UNIVERSITY OF CALIFORNIA, SAN DIEGO

Internet traffic measurement:
What's going on in my network?

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Cristian Estan

Committee in charge:

George Varghese, Chairperson
Stefan Savage
Vern Paxson
Fan Chung Graham
Keith Marzullo
Rene L. Cruz

2003

Copyright

Cristian Estan, 2003

All rights reserved.

The dissertation of Cristian Estan is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2003

TABLE OF CONTENTS

Signature Page . . . iii

Table of Contents . . . iv

List of Figures . . . viii

List of Tables . . . xii

Acknowledgments . . . xiv

Vita and Publications . . . xv

Abstract . . . xvi

I Introduction . . . 1
A. Traffic measurement challenges . . . 2
B. Directions for improving traffic measurement . . . 4
C. Dissertation outline . . . 5

II Traffic Measurement in the Internet Today . . . 6
A. Traffic measurement needs . . . 7
B. SNMP counters . . . 8
C. Flow level measurement . . . 8
D. Traffic summaries . . . 10
E. A wide definition of traffic measurement . . . 10
F. Chapter summary . . . 12

III Identifying Large Flows of Traffic . . . 14
A. Motivation . . . 15
B. Problem definition . . . 17
C. Related work . . . 18
D. Our solution . . . 20
1. Sample and hold . . . 21
2. Multistage filters . . . 24
3. Improvements to the basic algorithms . . . 27
E. Analytical evaluation of our algorithms . . . 30
1. Sample and hold . . . 30
2. Multistage filters . . . 34
F. Comparing Measurement Methods . . . 36
1. Comparison of the core algorithms . . . 37
2. Comparing Measurement Devices . . . 39
G. Dimensioning traffic measurement devices . . . 42
1. Dimensioning the multistage filter . . . 43
H. Measurements . . . 44
1. Comparing Theory and Practice . . . 46

2. Evaluation of complete traffic measurement devices 50
I. Implementation Issues 51
J. Acknowledgements 53
K. Chapter summary 53

IV Bitmap Algorithms for Counting Flows 55
A. Introduction 56
1. Problem Statement 57
2. Motivation 58
B. Related work 61
C. A family of counting algorithms 63
1. Direct bitmap 64
2. Virtual bitmap 65
3. Multiresolution bitmap 66
4. Triggered bitmap 68
5. Flow sample and hold 69
6. Handling packet removals 71
7. Combining or comparing multiple measurement intervals 71
D. Algorithm Analysis 72
1. Estimate Formulae 72
2. Accuracy 73
3. Configuring the bitmaps 76
E. Measurement results 79
1. Virtual bitmap 80
2. Multiresolution bitmap 80
3. Triggered bitmap 83
F. Acknowledgments 85
G. Chapter summary 86

V Explicit, Concise Description of Traffic Mixes with Traffic Clusters 88
A. Introduction 89
B. Multi-dimensional Traffic Clusters 90
1. Defining Traffic Clusters 92
2. Operations on Traffic Clusters 94
C. Algorithms 95
1. Unidimensional clustering 97
2. Multidimensional clustering 102
D. Related work 109
E. Future work: traffic synopses 109
1. Introduction 110
2. Problem definition 110
3. Packet synopses 111
4. Byte synopses 114
5. Flow synopsis 115
6. Summary 117
F. Acknowledgments 117
G. Chapter summary 118

VI Adapting Traffic Measurement to the Traffic Mix . . . 119
A. Inter-interval adaptation . . . 120
1. Adapting the threshold for algorithms for identifying large flows . . . 120
2. Calibrating the threshold adaptation algorithm . . . 121
3. Adaptive bitmap . . . 124
B. Intra-interval adaptation . . . 128
C. Acknowledgments . . . 134
D. Chapter summary . . . 134

VII AutoFocus: a Flexible Traffic Analysis System . . . 136
A. System description . . . 136
B. Experience with AutoFocus . . . 139
1. Comparison to unidimensional methods . . . 139
2. Experience with analysis of traffic traces . . . 141
C. Acknowledgments . . . 145
D. Chapter summary . . . 145

VIII Conclusions . . . 146
A. Summary of dissertation . . . 146
B. Future directions . . . 150

Appendix A Cisco NetFlow . . . 154
1. Basic NetFlow . . . 154
2. NetFlow Aggregation . . . 155
3. Sampled NetFlow . . . 155
4. The accuracy of sampled NetFlow . . . 156
5. The memory requirements of sampled NetFlow . . . 157
6. Keeping a queue of packet headers . . . 157

Appendix B Analysis details for algorithms for identifying large flows . . . 159
1. Details of the analysis for multistage filters . . . 159
2. Analysis of algorithms for identifying large flows when flow sizes have a Zipf distribution . . . 164
2.1. Sample and hold with a Zipf distribution of flow sizes . . . 164
2.2. Multistage filters with a Zipf distribution of flow sizes . . . 167

Appendix C Defining large flows with leaky buckets . . . 170
1. Analytical evaluation of the parallel multistage filter using leaky buckets . 171
2. Implementing multistage filters with leaky buckets . . . 177

Appendix D Measurements of algorithms for idenifying large flows . . . 178
1. Measurements of sample and hold . . . 178
1.1. Comparing the behavior of the basic algorithm to the analytic results 178
1.2. The effect of preserving entries . . . 182
1.3. The effect of early removal . . . 183
2. Measurements of Multistage filters . . . 184
2.1. Comparing the behavior of basic filters to the analytic results . . . 184

2.2. The effect of conservative update 186
2.3. The effect of preserving entries and shielding 187

Appendix E Multiresolution bitmap details 190
1. Analysis of average error for the virtual and multiresolution bitmaps . . . 190
2. Configuring the multiresolution bitmap 194
3. Multi-resolution bitmap versus probabilistic counting 195
4. Hardware implementation of multiresolution bitmap 196

Appendix F Measurements of the error of direct and virtual bitmap 202

Appendix G The size of reports using traffic clusters 206
1. Comparison of actual report sizes and theoretical bounds for different threshold values 206
2. The effect of the length of the measurement interval 209
3. The effect of traffic diversity 210

Bibliography 211

LIST OF FIGURES

I.1 A traffic measurement system consists of components that produce progressively more refined representations of the observed traffic. At the various stages communication bandwidth, memory and processing power limitations can turn into bottlenecks 3
I.2 Various descriptions of the traffic differ in the amount of information they contain and in their conciseness. Descriptions in use today are at different points on the curve of tradeoffs between information content and conciseness. Progress means pushing this curve towards descriptions that are more informative and at the same time more concise. 4

III.1 The leftmost packet with flow label $F1$ arrives first at the router. After an entry is created for a flow (solid line) the counter is updated for all its packets (dotted lines) 22
III.2 Sampled NetFlow counts only sampled packets; sample and hold counts all packets after entry created 23
III.3 In a parallel multistage filter, a packet with a flow ID F is hashed using hash function $h1$ into a Stage 1 table, $h2$ into a Stage 2 table, etc. Each table entry contains a counter that is incremented by the packet size. If *all* the hashed counters are above the threshold (shown bolded), F is passed to the flow memory for individual observation. 25
III.4 In a serial multistage filter, a packet with a flow ID F is hashed using hash function $h1$ into a Stage 1 table. If the counter is below the stage threshold T/d, it is incremented. If the counter reaches the stage threshold the packet is hashed using function $h2$ to a Stage 2 counter, etc. If the packet passes all stages, an entry is created for F in the flow memory. 27
III.5 Conservative update: without conservative update (left), all counters are increased by the size of the incoming packet; with conservative update (right), no counter is increased to more than the size of the smallest counter plus the size of the packet 29
III.6 Cumulative distribution of flow sizes for various traces and flow definitions 46
III.7 Filter performance for a stage strength of k=3 49

IV.1 The flow count provided by Dave Plonka's FlowScan is used to detect denial of service attacks. 59
IV.2 The multiresolution bitmap from this example uses a single 7-bit hash function to decide which bit to map a flow to. It gives results no less accurate than the 3 virtual bitmaps, thus covering a wide range for the number of flows, but it performs a single memory update per packet. Note that all the unfilled "tiles" from these bitmaps, despite their different sizes, represent one bit of memory. 67
IV.3 Algorithm for computing the estimate of the number of active flows for a multiresolution bitmap. We first pick the base component that gives the best accuracy then add together the estimates for the number of flows hashing to it and all higher resolution components and finally extrapolate. 74

IV.4 When the flow density is too low, the "sampling error" takes over; when it is too high, "collision error" is the main factor. We get the best accuracy for a flow density of around $\rho = 1.6$. The estimate from Equation IV.4 matches well the experimental results, being slightly conservative (larger). See Section IV.E.1 for details on the experiment that produced this result. 75
IV.5 Configured for an average error of 10% 81
IV.6 Configured for an average error of 3% 82
IV.7 Configured for an average error of 1% 82

V.1 Each individual IP address sending traffic appears as a leaf. The traffic of an internal node is the sum of the traffic of its children. Nodes whose traffic is above H=100 (double circles) are the high volume traffic clusters. The Web server 10.8.0.8 is a large cluster in itself. While no individual DHCP address is large enough, their aggregate 10.8.0.0/29 is, so it is listed as a large cluster. 98
V.2 The clusters from the compressed report are represented with double circles. Node 10.8.0.8/31 is not in the compressed report because its traffic is exactly the sum of the traffic of its children. Node 10.8.0.8/30 is not in the compressed report because its traffic is within a small amount (35) of as what we can compute based on its two descendants in the report. 100
V.3 The multidimensional model combines unidimensional hierarchies (trees) into a graph. The hierarchy on the near side of the cube breaks up the traffic by prefixes; the hierarchy on the right side of the cube by protocol. For example, the node labeled C on the near side represents the Computer Science Department, the node labeled U on the right side represents the UDP traffic and the node labeled UC in the graph represents the UDP traffic of the Computer Science Department. 103
V.4 The algorithm for compressing traffic reports traverses all clusters starting with the more specific ones. The "estimate" counter of each cluster contains the total traffic of a set of non-overlapping more specific clusters that are in the compressed report. The clusters whose estimate is below their actual traffic by more than the threshold H , are included into the compressed report. 106

VI.1 Dynamic threshold adaptation to achieve target memory usage 120
VI.2 The large virtual bitmap replaces 6 of the components of the multiresolution bitmap. The size of the normal components is $b = 64$ bits and the size of the large virtual bitmap is $v = 1627$ bits. The adaptive bitmap guarantees an average error of at most 10% over the whole range, but if the number of flows falls into the "sweet spot" the average error can be as low as 3.1% 125
VI.3 Initialization of per-table PSH state at the beginning of each interval. . 130
VI.4 Algorithm for packet sample and hold on the src IP table with dynamically adapted sampling rate, applied to each packet whose src IP does not already exist in the src IP table. The algorithm works similarly on the other tables. 131

VI.5 Estimating the time it takes to fill up the other six eighths of memory based on the times it took to fill up the first two eighths (i.e. the two halves of the budget). 132
VI.6 Using a linear prediction that assumes that the rate at which entries will be created is the same as the rate at which they were created during the current budget underestimates the time it takes to fill the memory. Accounting for the fact that it takes progressively longer to fill up the memory as we advance in time gives a much better prediction. 133

VII.1 The report for the 17th of December 2002 (one of the 31 daily reports for this trace) contains compressed unidimensional reports on all 5 fields and the compressed multidimensional cluster report using a threshold of 5% of the total traffic. In the unidimensional reports the percentages indicate the share of the total traffic the given cluster has. In the multidimensional report they indicate the unexpectedness score. Note how much smaller the delta report is than the full report. 137
VII.2 Multidimensional report for Friday the 20th and Saturday the 21st of December 2002 using traffic clusters 140
VII.3 Unidimensional report for source network 140
VII.4 Unidimensional report for source port 140
VII.5 Unidimensional report for protocol 140
VII.6 Unidimensional report for destination network 140
VII.7 Unidimensional report for destination port 140
VII.8 The Sapphire/SQL Slammer worm shows up in the time series plots as a big increase in the traffic of the "Other" category around 21:30. Once we highlight the worm by putting it into a separate category, it is evident that while its traffic is significantly reduced at 22:10 when the infected internal hosts were neutralized, worm traffic persists at a lower level because of outside hosts spreading it into this network. 142

VIII.1 The algorithmic solutions to the building blocks we identified help relieve potential bottlenecks in the traffic measurement process 147

B.1 Algorithm for computing a tight high probability bound on the number of flows passing a parallel filter 165

D.1 Percentage of false negatives as the oversampling changes 179
D.2 Average error in the traffic estimates for large flows (without corrections to account for missed packets) 179
D.3 Maximum number of flow memory entries used 180
D.4 Average error in the traffic estimates for large flows (without corrections to account for missed packets) 181
D.5 Effect of early removal on memory usage 183
D.6 Actual performance for a stage strength of k=1 184
D.7 Actual performance for a stage strength of k=3 185
D.8 Conservative update for a stage strength of k=1 186
D.9 Conservative update for a stage strength of k=3 187

D.10 Change in memory usage due to preserving entries and shielding 188

E.1 The algorithm for numerically computing the best configuration for a multi-resolution bitmap considers four configurations for each value of k and returns the one using the least memory. For all software configurations we tested, the algorithm chose the third (line 21) or the fourth (line 29) configuration. For all hardware configurations it chose the third. 199
E.2 Probabilistic counting groups the bits horizontally into bitmaps that contain bits with different probabilities of being set, while multiresolution bitmaps group bits vertically with bits with the same probability of being set in the same component 200
E.3 Hardware for selecting the bit to be set 201

F.1 Average relative error of a direct bitmap with 1000 bits 203
F.2 Average relative error of a virtual bitmap with 1000 bits 203
F.3 Direct bitmap with 100 bits 204
F.4 Average relative error of a virtual bitmap with 100 bits 204
F.5 Bias for the 1000 bit bitmaps 205
F.6 Bias for the 100 bit bitmaps 205

G.1 The size of compressed traffic reports is proportional to the inverse of the threshold. For the same configuration, most of the traffic reports have very close sizes, except a few that are much smaller due to high volume individual connections that dominate the traffic mix (and push the threshold up). 208

LIST OF TABLES

II.1 Existing traffic measurement solutions have strengths (indicated by many stars) and weaknesses. The third column evaluates the diversity of the analyses supported by the output of the given system, the fourth looks at the human effort required to configure a given solution and the fifth at the information content of the results (including accuracy). 12

III.1 Comparison of the core algorithms: sample and hold provides most accurate results while pure sampling has very few memory accesses 37
III.2 Comparison of traffic measurement devices 41
III.3 The traces used for our measurements 45
III.4 Summary of sample and hold measurements for a threshold of 0.025% and an oversampling of 4 47
III.5 Comparison of traffic measurement devices with flow IDs defined by 5-tuple 52
III.6 Comparison of traffic measurement devices with flow IDs defined by destination IP 52
III.7 Comparison of traffic measurement devices with flow IDs defined by the source and destination AS 52

IV.1 The size of the direct bitmap scales sublinearly with N, roughly as $N/\ln(N\epsilon^2+1)$; the size for the virtual bitmap is proportional to the inverse of the square of the average error; and the size of the multiresolution bitmap scales with the logarithm of the number of flows over the square of the average error. 77
IV.2 The operating range of the components of the multiresolution bitmap is between ρ_{min} and ρ_{max}. The coefficient and the desired accuracy determine the size of the components $b = f(k)/\epsilon^2$. The larger the ratio between the resolutions of neighboring components k, the wider the range covered by a single component and the larger the component. . . . 78
IV.3 The traces used for our measurements 79
IV.4 The memory usage of port scan detection algorithms (Kbytes) 84
IV.5 The family of bitmap counting algorithms: each algorithm is best suited for a different setting. 84

VI.1 The average to maximum memory usage ratios for various configurations 122
VI.2 The range of measured adjustment ratios 123
VI.3 As we increase the number of components r replaced by the virtual bitmap, the size of the virtual bitmap v almost doubles for each new component replaced. The ratio between the average error of the large virtual bitmap and the multiresolution bitmap also increases exponentially, but at a slower rate than the size of the virtual bitmap. 127
VI.4 Comparison of adaptive bitmap and probabilistic counting errors, each using 16Kbits of memory 127

VIII.1 Traffic measurement solutions have strengths (indicated by many stars) and weaknesses. The first column evaluates the diversity of the analyses supported by the output of the given system, the second looks at the human effort required to configure a given solution and the third at the information content of the results (including accuracy). 147
VIII.2 Summary of contributions of this dissertation 151

D.1 Various measures of performance when using an early removal threshold of 15% of the threshold as compared with the values without early removal 182
D.2 Average error when preserving entries as percentage of the average error in the base case 188

G.1 Compressed reports are much shorter than uncompressed reports. Actual reports are orders of magnitude smaller than the theoretical bounds. 207
G.2 The size of compressed reports is slightly smaller for shorter measurement intervals. The number of active flows is an indication of the number records NetFlow would generate. 207
G.3 The greater diversity of backbone traffic (trace3) did not lead to significant increases in the size of the traffic reports. 208

Acknowledgments

There are many people to whom I am thankful for being able to write this dissertation. First of all I thank my family. My parents instilled a thirst for knowledge into me. My wife understood and supported me in many ways through the years it took to finish the work presented here. I thank professor *Kálmán Pusztai* who taught the first computer networking class I took and encouraged me to pursue this field and also to take my first real job as network administrator, a job I learned a lot from. I thank *S. Keshav* who was my first advisor in a Ph.D. program. He introduced me to the cutting edge in networking research and to the exciting life of a Silicon Valley startup and helped me find this wonderful place to pursue my graduate degree. I thank all my colleagues here at UCSD for a warm and stimulating environment. I especially thank *Florin Baboescu*, *Sriram Ramabhadran* and *Ranjita Bhagavan* for discussions that helped improve my papers and my coauthors, *Mike Fisk*, *Ken Keys*, *David Moore* and *Sumeet Singh*. I thank all of CAIDA and *K. Claffy* in particular for their support throughout these years. I thank the entire faculty of the Computer Science and Engineering Department for the opportunity to learn and broaden my horizons. I especially thank *Fan Chung Graham*, *Geoff Voelker* and *Alex Snoeren* for many valuable discussion about research and other topics as well. I thank *Vern Paxson* for his generous support and guidance. I thank *Stefan Savage* who was as a second advisor to me. The depth and breadth of his knowledge of the field always impressed me and I learned more from him than from anyone else except my advisor, *George Varghese*. He taught me a lot about how the network works and grows. He showed me that there are research topics both beautiful and of practical significance. Many of the things that I value in him cannot be taught: his sparkling creativity, his never ending enthusiasm, his wisdom of doing the right things at the right time and approaching a problem from the right angle with the right attitude. If at least some of these rubbed off on me, I'm sure I'll be a happy and successful man.

My work was supported by the Sensilla Project sponsored by NIST grant 60NANB1D0118, and by NSF Grant ANI 0074004.

VITA

September 27, 1971	Born, Satu Mare, Romania
1995	Engineer (B.A.) Computer Science Technical University of Cluj-Napoca, Romania
1996	M.Sc. Computer Science Technical University of Cluj-Napoca, Romania
1995–1998	Network administrator Romanian Educational Network
1998–1999	Research and teaching assistant Computer Science Department, Cornell University
1999–2000	Software developer, Ensim Corporation Sunnyvale, California
2000–2003	Research and teaching assistant Department of Computer Science and Engineering University of California, San Diego
2003	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

Cristian Estan, George Varghese and Mike Fisk. Bitmap Algorithms for Counting Active Flows on High Speed Links. In *Proceedings of the Internet Measurement Conference*, October 2003.

Cristian Estan, Stefan Savage and George Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proceedings of the ACM SIGCOMM*, August 2003.

Cristian Estan and George Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. In ACM Transactions on Computer Systems, August 2003.

ABSTRACT OF THE DISSERTATION

Internet traffic measurement:
What's going on in my network?

by

Cristian Estan
Doctor of Philosophy in Computer Science
University of California, San Diego, 2003
Professor George Varghese, Chair

One of the main reasons for the success of the Internet is its service model that emphasizes flexibility: any computer can send any type of data to any other computer at any time. While this freedom enabled the unhindered deployment of the applications popular today such as email and the world wide web, it also made the job of administering the network harder. In the phone network, exhaustive call logs are readily available, whereas the Internet lacks built-in traffic measurement features that would help the operators figure out how the network is used or misused. Existing solutions either lack the necessary detail, do not scale up to the speeds of today's networks, fail to extract the meaningful information from raw data or are not flexible enough to keep up with the ever changing traffic mix. This dissertation addresses these problems.

Traffic measurement is not driven by a single concrete goal. There is a large number of systems that perform traffic measurement to answer a wide variety of questions. The strategy adopted by this dissertation is to isolate tasks common to many traffic measurement systems and provide efficient algorithmic solutions to those tasks that can be used as building blocks. The four important building blocks addressed are: identifying and measuring large flows of traffic, counting flows, network traffic synopses and explicitly describing complex traffic mixes. The dissertation also discusses how to improve performance by adapting the parameters of these building blocks to the observed traffic. It concludes with the description of a complete traffic analysis system using many of these algorithmic solutions.

Chapter I

Introduction

All around the world millions of people use computer networks every day for email, casual web browsing, shopping, banking, telecommuting and voice or video calls. Businesses use them to link distant offices, transfer critical information in a timely manner and offer comprehensive and up to date information to their partners, clients and prospective customers. Governments and non-profit organizations benefit extensively from this cheap and efficient communication medium. As our reliance on computer networks increases so does the importance of them working reliably. Measuring and understanding data traffic is essential for ensuring the reliable operation and smooth growth of computer networks. The contribution of this dissertation is to describe new methods for measuring and analyzing the traffic of computer networks.

The Internet is the undisputed champion of computer networks today. Its success is in great part due to its service model that emphasizes flexibility: any computer can send any type of traffic to any other computer at any time. This freedom allowed new applications to flourish unhindered by unnecessary constraints. For example, the most popular application today, the web, was developed long after the technical foundations of the Internet were firmly in place.

One cost we pay for the flexibility offered by the core protocols of the Internet is the difficulty of monitoring how the network is actually used. In the telephone network, exhaustive logs are available that describe the relevant details of all the calls handled by any given switch. Not only are these types of call logs most often not available for the Internet, but even when we can obtain them, the diversity and heterogeneity of Internet

"calls" goes far beyond the variability we see in telephone conversations.

Another problem with the flexibility and permissiveness of the network is that it offers ample opportunities for misuse by malicious parties. Worms, viruses, attacks and intrusions are unfortunately nearly ubiquitous in today's Internet. These undesirable phenomena make the measurement and monitoring of Internet traffic even more important.

I.A Traffic measurement challenges

Besides the difficulties caused by the flexibility of the Internet service model, measurement systems must pass two more hurdles caused by the volume of the traffic: huge amounts of raw data and ever increasing speeds at which data arrives. As the volume of traffic and the speed of the network keep increasing exponentially, these problems quickly aggravate, so any traffic measurement system meant to last must not only be able to handle today's speeds, but also scale well into the future.

Given the importance of Internet traffic measurement and analysis there is a large body of scientific research dealing with the subject and there are multiple systems deployed today that achieve a wide variety of traffic measurement goals. Despite the variety of current solutions, because of the difficulties outlined above, there are many important measurement tasks that no system can perform well enough. Existing solutions either lack the necessary detail, do not scale to high enough speeds, fail to extract the meaningful information from raw data or are not flexible enough to keep up with the ever changing traffic mix. This dissertation addresses each of these shortcomings.

Figure I.1 presents the general structure of a traffic measurement system. There is a measurement device that can be standalone equipment or be a module of a router that captures the traffic of a network link and performs some preprocessing. The management station collects raw data from measurement devices throughout the network and converts it into traffic summaries for the network administrator. Many actual traffic measurement systems do not match this structure exactly. Some are standalone "boxes" that are deployed in the network to perform their task (e.g. network intrusion detection systems), others combine more layers of distributed components working together.

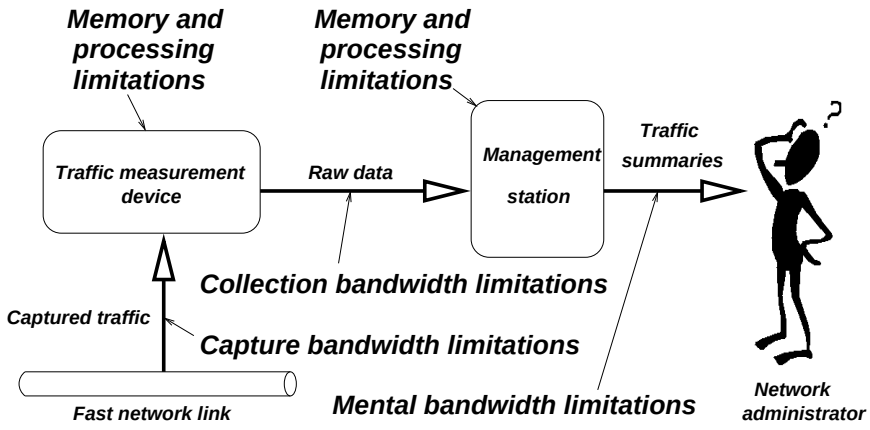


Figure I.1: A traffic measurement system consists of components that produce progressively more refined representations of the observed traffic. At the various stages communication bandwidth, memory and processing power limitations can turn into bottlenecks

Yet, fundamentally they have a common structure. At one end there are one or more components that observe the traffic and at the other end the final component delivers some useful information to a human observing the network or an automated mechanism that takes action based on the observed traffic. All the components on this path perform the same job: they transform their input (which is a description of the traffic) into a more compact description trying to maintain as much as possible of the useful information.

The reason why we prefer concise descriptions is pragmatic: memory and communication bandwidth can become bottlenecks at all stages. Limits on processing power are sometimes very strict and they make the task of refining the traffic descriptions even more challenging. The speed of large and cheap DRAM memories increases only by 9% per year [PH98] while the speed of the links increases at 50% per year [R00]. DRAM speeds are already behind line speeds and the gap is going to increase only, so traffic measurement components that look at every packet are going to be relegated more and more to hardware where memory (SRAM) and processing are scarce resources. In the final stages of the traffic measurement system the network administrator himself can become a bottleneck: the amount of information he can absorb is limited, hence the

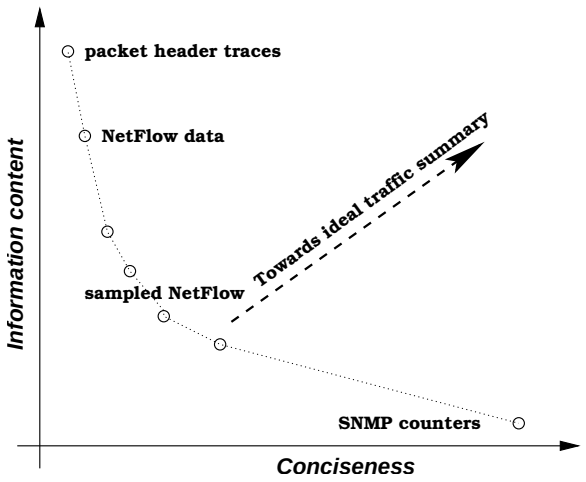


Figure I.2: Various descriptions of the traffic differ in the amount of information they contain and in their conciseness. Descriptions in use today are at different points on the curve of tradeoffs between information content and conciseness. Progress means pushing this curve towards descriptions that are more informative and at the same time more concise.

mental bandwidth limitation imposed onto traffic reports.

I.B Directions for improving traffic measurement

There are many measures that define a good component of a traffic measurement system. Obviously, the less memory and processing it requires, the better. Also the more concise the output, the less likely it is to turn communication bandwidth limitations into bottlenecks. Usually there is a tradeoff between the conciseness of the output and the amount of information it preserves about the traffic mix, which can be often quantified through some measure of accuracy. Figure I.2 shows that various descriptions of the traffic mix in use today have different tradeoffs between information content and conciseness. Better solutions can give us descriptions of the traffic which are both more informative and more concise. This gives us another criterion for judging the quality of processing blocks.

I.C Dissertation outline

Traffic measurement systems differ not only in the exact details of their structure, but also in the concrete questions they are meant to answer. To give a better perspective on the contributions of this dissertation, Chapter II gives a succinct overview of traffic measurement problems and solutions in the Internet today. Instead of selecting some concrete measurement problems to solve, we[1] identify tasks common to many traffic measurement systems and proposes algorithmic solutions with quantifiable performance that can be used a building blocks in various systems.

Chapter III presents algorithms for efficiently identifying large flows of traffic. Chapter IV presents bitmap algorithms for counting the number of active flows. The moderate memory and processing power required by the solutions from these two chapters makes them very suitable for hardware solutions that handle high volume of data at the early stages of a traffic measurement system. Chapter V describes algorithms for explicitly summarizing traffic mixes using traffic clusters. The processing involved and the focus on making the summaries explicit and readable by humans makes this building block best suited for the final stages of the system.

Often the processing blocks have a fixed amount of resources (memory, CPU power) that is not shared with other components. The degree to which these resources are used often varies significantly with the traffic mix. In these cases the processing blocks can improve the accuracy of their results by adjusting some configuration parameters dynamically based on the traffic mix they see. This is the topic we explore in Chapter VI. Chapter VII presents a complete traffic analysis system based on traffic clusters that provides generous support to the network operator for understanding the traffic mix and identifying misuses of network resources. The final chapter presents the conclusions of this dissertation.

[1] Every single section, subsection, figure, table, equation and proof in this dissertation is my work more than anyone else's. However, I did not work alone, I received generous guidance and advice from my adviser and other members of my committee. While the work presented here was published in papers that I was first author on, I was not the only author. Therefore, throughout the dissertation I will use first person plural when referring to the authors.

Chapter II

Traffic Measurement in the Internet Today

The Internet is a confederacy of networks owned by various organizations all around the globe. These networks have two important things in common: they all run the same network protocol (they "speak the same language"), IP, and they work together to ensure that all networks can communicate with each other. The IP protocol mandates that communication between computers take place through chunks of data called packets that are handled by the Internet separately. Besides the useful data, packets have some overhead too: a header that consists of fields that make it possible for the network to deliver the packets to their destination. The networks consist of nodes called routers connected by links. For each packet they receive, routers examine the destination IP address and a few other fields in the packet header to decide which link to send it on so that it makes progress towards the destination.

It is useful to distinguish between two types of networks: edge networks and transit networks. Edge networks are confined to a small geographical area, like a campus, and their purpose is to connect the computers of a given organization to each other and to the rest of the Internet. These networks are usually built around LAN (Local Area Network) technologies such as the Ethernet family of protocols, and they are overprovisioned: the traffic between computers within the same network seldom approaches the capacity of the network. Edge networks connect to the rest of the Internet through a

small number of lower speed links that are typically not overprovisioned with respect to peak traffic. Transit networks are owned by ISPs (Internet Service Providers) and their purpose is to connect the networks of the ISPs' customers. The most important transit networks are backbones operated by major ISPs that have routers connected by high speed links in many locations across the globe. The costs of the long distance high speed links and those of the high speed routers that make up the backbone are very high, so vastly overprovisioning them is not economically feasible.

II.A Traffic measurement needs

There are four main uses for traffic measurement in the Internet. The owners of networks use traffic measurement to make decisions about the growth of the network (e.g. which links need to be upgraded). ISPs use traffic measurement for billing customers based on their traffic. Network administrators look at the traffic to detect malicious activities. Researchers, network operators and equipment manufacturers turn to traffic measurement to monitor the functioning of various protocols used in the network. There are two main methods of obtaining measurement data: from the networking equipment that handles the traffic and from specialized devices whose purpose is to capture measurement data. Especially when the raw data is provided by networking devices, it needs to be further processed, and this is usually done in software on general purpose computers.

The field of Internet measurement contains another salient sub-field, active measurement. Unlike passive measurement, which records data about the traffic on the network, active measurement sends out probe packets and measures how they, and/or their replies, traverse the network. Active measurement is typically used to measure properties of the network: packet delays, packet loss rates, jitter, bandwidth, symmetry, stability [P97], and topology [SMW02]. While those metrics are influenced at various timescales by the production traffic, we know of no significant attempt to use active measurement to measure production traffic, thus we will not discuss it any further in this dissertation.

II.B SNMP counters

SNMP counters [MR91] are a very basic and widely available traffic measurement mechanism available in all routers today. Among other things, they count the number of packets and bytes transmitted on each of the links of a router. There are many traffic measurement applications that use them. The most common one is MRTG [OR95]: it collects the values of the counters, and produces web pages with time series plots of the traffic. Another important use of these counters is in billing: some ISPs charge their clients based on the total traffic they send and receive on their link to the ISP.

While the information offered by the SNMP counters is useful, it omits a lot of detail: they do not say what type of traffic is using the links. For example let's assume that the backbone link of Austin to Atlanta has too much traffic. If most of this traffic originates from Chicago and goes to Florida, a new link from Chicago to Washington D.C. might help, whereas if the bulk of the traffic is from Houston to Atlanta, it probably won't. More generally, for network planning, ISPs need to know the traffic matrix: the traffic between pairs of clients or pairs of traffic aggregation locations for the network (POPs). There are many approaches for reconstructing the traffic matrix based on SNMP counters and network topology [MTSBD02] and they are collectively known as network tomography. One disadvantage of these methods is that they can only give approximate traffic matrices.

II.C Flow level measurement

Flow level measurement provides traffic information with more detail. In the strict sense of the word, "flows" in the Internet are the equivalent to calls in the phone network: they define a conversation between two endpoints and are identified by certain values for 5 specific fields in the headers of the packets that make up the flow. The five fields are source IP address, destination IP address, protocol number, source port and destination port. The last three fields indicate which application generated the flow (web, email, etc.). Besides these fields, a flow record contains counters for the number of packets and bytes in the flow, timestamps for when the first and last packets were received, information about protocol flags and a few other details. This approach is

standardized by the Real-Time Flow Measurement (RTFM) [BMR99] Working Group of the IETF. Today's routers have features such as Cisco's NetFlow [CN] or LFAP [RL] that give flow level information about traffic. The flow level information can be used to compute accurate traffic matrices[FGLR+00]. Flow level records can also be used for billing and accounting that differentiates between types of traffic (e.g. traffic that stays within the ISP can be billed at a lower rate than traffic that needs to use an expensive link between continents).

The main problem with the flow measurement approach is its lack of *scalability*. Measurements on MCI OC-3 backbone traces as early as 1997 [TMW97] showed over 250,000 concurrent flows. More recent measurements in [FP99] using a variety of traces show the number of flows between end host pairs in a one hour period to be as high as 1.7 million. It can be feasible for flow measurement devices to keep up with the increases in the number of flows only if they use the cheapest memories: DRAMs. Updating per-packet counters in DRAM is already impossible with today's line speeds: the router has 32 nanoseconds to process a 40 byte packet at OC192 speeds (10 gigabits per second). Further, the gap between DRAM speeds (improving 7-9% per year) and link speeds (improving 100% per year[R00]) is steadily increasing. Cisco NetFlow, which keeps its flow counters in DRAM, solves this problem by sampling [CSN]: only sampled packets result in updates to the flow records. Appendix A has a more detailed description and analysis of NetFlow.

When very few packets are sampled, most flow records contain a single packet; thus, looking up the flow record for each packet is a waste of resources. sFlow [PPM01] is a simpler alternative that delivers to the collection station the headers of sampled packets instead of flow records. Both Sampled NetFlow and sFlow have lower measurement accuracy than NetFlow because of the statistical errors introduced by sampling. Another problem is that even with sampling the amount of raw data generated by NetFlow can overwhelm the collection station or its network connection. One can reduce this collection overhead by using one of NetFlow's aggregation schemes. For example NetFlow can aggregate records by destination prefix (roughly equivalent to destination network) instead of reporting each flow individually. But aggregation loses information. If reports are aggregated by destination prefix we cannot recover source prefix information or

individual destination addresses. Flow sampling [DLT01] has been proposed to reduce the amount of flow data stored by sampling flow records with probability dependent on flow sizes. By doing this it can reduce the memory needed to store flow records without significantly impacting the accuracy of the results of a wide range of analyses.

A more accurate method of collecting raw data about the traffic is to use packet header traces. Traces are usually collected by general purpose computers using software such as tcpdump[T] or with specialized hardware such as DAG cards[D] that works with higher speed link technologies typically used in the backbones. The amount of data generated by packet header capture is often a problem, so this method is not suited for ubiquitous deployment throughout the network. Sampling and filtering can reduce the amount of data generated. Traces are very often used by researchers because of the wealth of information they provide.

II.D Traffic summaries

There are many software applications such as CoralReef [C] or the IPMON project [I] that produce various breakdowns of traffic based on packet traces. For example, the list of top k port numbers (those responsible for most traffic) gives a good summary of the applications that generate most traffic.

A visualization tool called FlowScan [P00] processes NetFlow records. It produces informative time series plots of the traffic by breaking it down into different categories based on application, the IP prefixes associated with local networks or other fields in the flow records.

II.E A wide definition of traffic measurement

In a wider sense traffic measurement is part of the solution to many problems in computer networking that are traditionally considered to be outside the traffic measurement domain. The two examples discussed below are network intrusion detection systems and fair scheduling. Both of these examples include components that take action based on the results of the measurements and other components that perform specialized tasks, but since at least conceptually they have components that do traffic measurement,

they can benefit from solutions developed for traffic measurement.

Fair scheduling[DKS89] aims to ensure fairness among flows when the network is congested. The routers are supposed to drop packets of flows that are above their fair share of the link. Thus routers need to measure the rate of the flows that use the most bandwidth in order to decide how much to reduce their rate. Many existing schemes measure all flows and have scalability problems.

Network intrusion detection systems (IDS) such as Bro [P99] and Snort [R99] try to protect networks from intrusions and attacks coming from the outside. If a certain packet is determined to be malicious, the IDS[1] can drop it thus protecting the intended victim from harm. Many IDSes use vast repositories of signatures that match exploits that can be used to subvert or otherwise harm computers on the protected network. Besides the task of looking for the signatures of exploits there are a other things IDSes can do. For example they can detect port scans (the attacker probing to see what services are installed before trying to compromise them) by counting the number of connections open by individual outside addresses and raising an alarm if someone makes too many connection attempts. There are other forms of malicious traffic that today's IDSes cannot adequately protect us from: denial of service (DoS) attacks that work by flooding the victim with often fake requests or packets, network worms and viruses that propagate without human intervention. Better measurement methods might help detect these undesirable events more promptly and with higher confidence.

Widening our horizons beyond applications traditionally considered part of traffic measurement, does not only allow us to identify other systems relying on unusual forms of traffic measurement. It also lets us see how we can use for traffic measurement purposes systems whose original purpose is something else, ACLs (Access Control Lists) are a feature of routers that allows network operators to express complex classification rules that define traffic aggregates. While the purpose of ACLs is to implement fine grained access control policies, a corresponding counter can also be incremented for any packet that matches an ACL rule; thus, they can be used to measure many types of aggregates.

[1]Strictly speaking this system is not and IDS, but an inline Intrusion Prevention System (IPS), but we use the term IDS broadly.

Solution	Section	Flexible analyses	Automation	Info. content	Output size	Memory used	Readability
SNMP counters	II.B		***	*	****	****	*
MRTG plots	II.B		***	*	****	***	****
Tomography TM	II.B		***	**	***	***	**
tcpdump(traces)	II.C	****	***	*****		****	
NetFlow	II.C	***	***	****	*		
Sampled NetFlow	II.C	***	**	**	**	*	
Aggr. NetFlow	II.C	*	***	**	**		*
Flow sampling	II.C	***	**	***	***		
SFlow	II.C	***	***	**	*	****	
top K reports	II.D		***	**	****	*	****
FlowScan	II.D	*	**	**	****	**	****
ACLs	II.E	***		***	****	***	**

Table II.1: Existing traffic measurement solutions have strengths (indicated by many stars) and weaknesses. The third column evaluates the diversity of the analyses supported by the output of the given system, the fourth looks at the human effort required to configure a given solution and the fifth at the information content of the results (including accuracy).

II.F Chapter summary

This chapter surveyed much of the very diverse palette of existing traffic measurement solutions. The strengths and weaknesses of existing traffic measurement solutions are summarized in Table II.1. Strengths are indicated by many stars and weaknesses by few.

The third column looks at the diversity of the analyses supported by the output of the given method. SNMP counters, MRTG plots, traffic matrices and top K reports are fundamentally inflexible. Aggregated NetFlow and FlowScan offer limited flexibility by through their configuration parameters. ACLs offer more flexibility because as they can describe many types of traffic aggregates. NetFlow, sampled NetFlow, sFlow and flow sampling offer a different type of flexibility as their output is meant to be used by later analyses. Traces allow even more flexibility since they can enable analyses based on packet content or fine grained timestamps.

The fourth column looks at how automated the solutions are. ACLs are funda-

mentally not automated because the network administrator has to introduce manually definitions of the aggregates he is interested in. All the others are automated, but with sampled NetFlow, FlowScan and flow sampling need some configuration.

The fifth column tries to quantify the information content of the various representations of network traffic. Traces and NetFlow give the most detailed description. Flow sampling loses some of the accuracy of NetFlow, and sampled NetFlow and sFlow are even less accurate. Aggregated NetFlow, top K reports and FlowScan aggregate along the directions they are interested in and thus lose information about other dimensions. SNMP counters and MRTG reflect only information about the traffic volume, while traffic matrices based on tomography contain information about the composition of the traffic, but they are inaccurate. ACLs can be configured to capture much interesting information.

The sixth column describes the output size. Traces are huge. NetFlow and sFlow data are somewhat smaller. sampling and aggregation can reduce the size of NetFlow data, and flow sampling can be used to achieve even more significant reductions.

The seventh column describes the memory used by the various systems. SNMP counters, traces and sFlow need to hold little state other than some buffers. ACLs need slightly more memory to keep the rules and the counters. NetFlow uses vast amounts of memory due to the large number of flows and aggregated NetFlow and flow sampling also need to pay this cost. Sampled NetFlow uses somewhat less memory as it sees fewer flows. The usual methods for computing top K reports rely on keeping large hash tables in memory.

The last column shows the readability of the output of various systems. Those whose output was meant for human consumption such as MRTG, top K reports and FlowScan fare well while the others don't.

Common weaknesses to many of the existing solutions are that their results lack the necessary detail, they do not scale to high enough speeds because of their memory usage, and they fail to extract the meaningful information from raw data or are not flexible enough to keep up with the ever changing traffic mix. This rest of this dissertation addresses each of these shortcomings.

Chapter III

Identifying Large Flows of Traffic

One of the most useful types of traffic reports are the "top K" reports that identify the largest flows of traffic, which are important due to their impact on the network. These reports are usually computed offline, based on large amounts of raw measurement data. A natural way to reduce resource consumption throughout the traffic measurement system (Figure I.1) is to identify and track the large flows early.

This chapter describes two novel and scalable algorithms for identifying large flows: *sample and hold* and *multistage filters*, which take a constant number of memory references per packet and use a small amount of memory. If M is the available memory, we show analytically that the errors of our new algorithms are proportional to $1/M$; by contrast, the error of an algorithm based on classical sampling is proportional to $1/\sqrt{M}$, thus providing much less accuracy for the same amount of memory. We also describe optimizations such as *early removal* and *conservative update* that further improve the accuracy of our algorithms, as measured on real traffic traces, by an order of magnitude.

The chapter is organized as follows. Section III.A motivates the need for a building block for traffic measurement systems that detects large flows. Section III.B gives the formal definition of what algorithms for identifying large flows have to do. We describe related work in Section III.C, describe our main ideas in Section III.D, and provide a theoretical analysis in Section III.E. We theoretically compare our algorithms with current solutions in Section III.F. After showing how to dimension our algorithms in Section III.G, we describe experimental evaluation on traces in Section III.H. We end with implementation issues in Section III.I.

III.A Motivation

> *If we're keeping per-flow state, we have a scaling problem, and we'll be tracking millions of ants to track a few elephants.* — Van Jacobson, End-to-end Research meeting, June 2000.

In the strict sense of the word, "flows" in the Internet define a conversation between two endpoints and are identified by certain values for 5 specific fields in the headers of the packets that make up the flow. In the more general sense used in this chapter, flows denote aggregates of traffic defined by values of specific header fields. Collecting flow records (in the strict sense of the word) is the standard approach for traffic measurement supported by routers through features such as NetFlow [CN]. The main problem with the flow measurement approach is its lack of *scalability*: there are too many flows in the backbone and link speeds are too high for DRAM memories that can accommodate large flow tables. Cisco NetFlow, which keeps its flow counters in DRAM, solves this problem by sampling: only sampled packets result in updates. But Sampled NetFlow has problems of its own (as we show later), since sampling affects measurement accuracy.

Despite the large number of flows, a common observation found in many measurement studies (e.g., [FGLR+00, FP99]) is that a small percentage of flows accounts for a large percentage of the traffic. [FP99] shows that for various backbone traces 9% of the flows between AS pairs account for 90% of the byte traffic between all AS pairs. For many applications, knowledge of these large flows is probably sufficient. [FP99, PPS01] suggest achieving scalable differentiated services by providing selective treatment only to a small number of large flows. [FGLR+00] underlines the importance of knowledge of "heavy hitters" for decisions about network upgrades and peering. [DLT01] proposes a usage sensitive billing scheme that relies on exact knowledge of the traffic of large flows but only samples of the traffic of small flows.

We conclude that it is infeasible to accurately measure all flows on high speed links, but many applications can benefit from accurately measuring only the few large flows. One can easily keep counters for a few large flows using a small amount of fast memory (SRAM). However, how does the device know which flows to track? If one keeps state for *all* flows to identify the *few* large flows, our purpose is defeated.

Our algorithms for identifying large flows can potentially be used to solve many problems. Since different applications define flows by different header fields, we need a separate instance of our algorithms for each of them. Applications we envisage include:

- **Scalable Threshold Accounting:** The two poles of pricing for network traffic are usage based (e.g., a price per byte for each flow) or duration based (e.g., a fixed price based on duration). While usage-based pricing [MV95, SCEH96] has been shown to improve overall utility, usage based pricing in its most complete form is not scalable because we cannot track all flows at high speeds. We suggest, instead, a scheme where we measure all aggregates that are above $z\%$ of the link; such traffic is subject to usage based pricing, while the remaining traffic is subject to duration based pricing. By varying z from 0 to 100, we can move from usage based pricing to duration based pricing. More importantly, for reasonably small values of z (say 1%) threshold accounting may offer a compromise between that is scalable and yet offers almost the same utility as usage based pricing. [AC01] offers experimental evidence based on the INDEX experiment that such threshold pricing could be attractive to both users and ISPs. [1]

- **Real-time Traffic Monitoring:** Many ISPs monitor backbones for hot-spots in order to identify large traffic aggregates that can be rerouted (using MPLS tunnels or routes through optical switches) to reduce congestion. Also, ISPs may consider sudden increases in the traffic sent to certain destinations (the victims) to indicate an ongoing attack. [MBFI+01] proposes a mechanism that reacts as soon as attacks are detected, but does not give a mechanism to detect ongoing attacks. For both traffic monitoring and attack detection, it may suffice to focus on large flows.

- **Scalable Queue Management:** At a smaller time scale, scheduling mechanisms seeking to approximate max-min fairness need to detect and penalize flows sending above their fair rate. Keeping per flow state only for these flows [FKSS01, PPS01] can improve fairness with small memory. We do not address this application further, except to note that our techniques may be useful for such problems. For

[1] Besides [AC01], a brief reference to a similar idea can be found in [SCEH96]. However, neither paper proposes a fast mechanism to implement the idea.

example, [PPS01] uses classical sampling techniques to estimate the sending rates of large flows. Given that our algorithms have better accuracy than classical sampling, it may be possible to provide increased fairness for the same amount of memory by applying our algorithms.

III.B Problem definition

A reasonable goal is to devise an algorithm that identifies large flows *using memory that is only a small constant larger than is needed to describe the large flows in the first place*. This is the central question addressed by this chapter. We present two algorithms that provably identify large flows using such a small amount of state. Further, our algorithms use only a few memory references per packet, making them suitable for use in high speed routers. Our algorithms produce more accurate estimates than Sampled NetFlow, but they do processing and access memory for each packet. Therefore the small amount of memory they use has to be fast memory operating at line speeds.

A flow is generically defined by an *identifier* (values for a set of specified header fields). We can also generalize by allowing the identifier to be a *function* of the header field values (e.g., using prefixes instead of addresses based on a mapping using route tables). Flow definitions vary with applications: for example, for a traffic matrix one could use identifiers defined by distinct source and destination network numbers. On the other hand, for identifying TCP denial of service attacks one could use the destination IP address as a flow identifier. Note that we do not require the algorithms to support simultaneously all these ways of aggregating packets into flows. The algorithms know a priori which flow definition to use and they do not need to ensure that a posteriori analyses based on different flow definitions are possible (as they are based on NetFlow data).

Large flows are defined as those that send more than a given threshold (say 0.1% of the link capacity) during a given measurement interval (1 second, 1 minute, or even 1 hour). Appendix C gives alternative definitions and algorithms based on defining large flows via leaky bucket descriptors.

An ideal algorithm reports, at the end of the measurement interval, the flow IDs and sizes of all flows that exceeded the threshold. A less ideal algorithm can fail in three ways: it can omit some large flows, it can wrongly add some small flows to the report, and can give an inaccurate estimate of the traffic of some large flows. We call the large flows that evade detection *false negatives*, and the small flows that are wrongly included *false positives*.

The minimum amount of memory required by an ideal algorithm is the inverse of the threshold; for example, there can be as many as 1000 flows that use more than 0.1% of the link. We will measure the performance of an algorithm by four metrics: first, its memory compared to that of an ideal algorithm; second, the algorithm's probability of false negatives; third, the algorithm's probability of false positives; and fourth, the expected error in traffic estimates.

III.C Related work

The primary tool used for flow level measurement by IP backbone operators is Cisco NetFlow [CN]. NetFlow keeps per flow counters in a large, slow DRAM. Basic NetFlow has two problems: **i) Processing Overhead:** updating the DRAM slows down the forwarding rate; **ii) Collection Overhead:** the amount of data generated by NetFlow can overwhelm the collection server or its network connection. For example [FGLR+00] reports loss rates of up to 90% using basic NetFlow.

The processing overhead can be alleviated using sampling: per-flow counters are incremented *only* for sampled packets[2]. Classical random sampling introduces considerable inaccuracy in the estimate; this is not a problem for measurements over long periods (errors average out) and if applications do not need exact data. However, we will show that sampling does not work well for applications that require true lower bounds on customer traffic (e.g., it may be infeasible to charge customers based on estimates that are *larger* than actual usage) and for applications that require accurate data at small time scales (e.g., billing systems that charge higher during congested periods).

The data collection overhead can be alleviated by having the router aggregate

[2]NetFlow performs 1 in N periodic sampling, but to simplify the analysis we assume in this chapter that it performs ordinary sampling processing each packet with probability 1/N independently.

flows (e.g., by source and destination AS numbers) as directed by a manager. However, [FP99] shows that even the number of aggregated flows is very large. Under unusual traffic mixes, the amount of data generated by NetFlow can be extreme. For example, collecting packet headers for Code Red traffic on a class A network [M01] produced 0.5 Gbytes per hour of compressed NetFlow data and aggregation reduced this data only by a factor of 4. Techniques described in [DLT01] can be used to reduce the collection overhead at the cost of further errors. However, it can considerably *simplify* router processing to only keep track of heavy-hitters (as proposed here) if that is what the application needs.

Many papers address the problem of mapping the traffic of large IP networks. [FGLR+00] deals with correlating measurements taken at various points to find spatial traffic distributions. [DG00] describes a mechanism for identifying packet trajectories in the backbone, that is not focused towards estimating the traffic between various networks. [SRS99] proposes that edge routers identify large long lived flows and route them along less loaded paths to achieve stable load balancing. Our algorithms might allow the detection of these candidates for re-routing in higher speed routers too.

Bloom filters [B70] and stochastic fair blue [FKSS01] use similar techniques to our parallel multistage filters to compute very different metrics (set membership and drop probability). In [TR99, SKR01] the authors look at various mechanisms for identifying the high rate flows to ensure quality of service. Their algorithms rely on caching flow identifiers and while some of their techniques are similar to our sampling technique and to what we call "preserving entries", their algorithms as a whole are quite different form ours. Gibbons and Matias [GM98] consider synopsis data structures that use small amounts of memory to approximately summarize large databases, but their algorithms have also been used for profiling program execution [BELV+00]. Their counting samples use the same core idea as our sample and hold algorithm. However, since the constraints and requirements in their setting (data warehouses updated constantly) are different from ours, the two final algorithms also differ. For example, we need to take into account packet lengths, we operate over a sequence of measurement intervals and our algorithms need to ensure low worst per packet processing times as opposed to amortized processing in the data warehouse context. In [FSGMU98], Fang et al look at efficient ways of

answering *iceberg queries*, or counting the number of appearances of popular items in a database. Their multi-stage algorithm is similar to multistage filters that we propose. However, they use sampling as a front end before the filter and use multiple passes. Thus, their final algorithms and analyses are very different from ours. For instance, their analysis is limited to Zipf distributions while our analysis holds for all traffic distributions. Cohen and Matias [CM03] independently discovered in the context of spectral Bloom filters the optimization to multistage filters we call "conservative update". In [KPS03] Karp et al. give an algorithm that is guaranteed to identify all elements that repeat frequently in a single pass. They use a second pass over the data to count exactly the number of occurrences of the frequent elements because the first pass does not guarantee accurate results. Building on our work, Narayanasamy et al. [NSSCV03] use multistage filters with conservative update to determine execution profiles in hardware and obtain promising results.

III.D Our solution

Because our algorithms use an amount of memory that is a constant factor larger than the (relatively small) number of large flows, our algorithms can be implemented using on-chip or off-chip SRAM to store flow state. We assume that at each packet arrival we can afford to look up a flow ID in the SRAM, update the counter(s) in the entry or allocate a new entry if there is no entry associated with the current packet.

The biggest problem is to identify the large flows. Two approaches suggest themselves. First, when a packet arrives with a flow ID not in the flow memory, we could make a place for the new flow by evicting the flow with the smallest measured traffic (i.e., smallest counter). While this works well on traces, it is possible to provide counter examples where a large flow is not measured because it keeps being expelled from the flow memory before its counter becomes large enough. This can happen even when using an LRU replacement policy as in [SKR01].

A second approach is to use classical random sampling. Random sampling (similar to sampled NetFlow except using a smaller amount of SRAM) provably identifies large flows. However, as the well known result from Table III.1 on 37 shows, random

sampling introduces a very high relative error in the measurement estimate that is proportional to $1/\sqrt{M}$, where M is the amount of SRAM used by the device. Thus, one needs very high amounts of memory to reduce the inaccuracy to acceptable levels.

The two most important contributions of this chapter are two new algorithms for identifying large flows: *Sample and Hold* (Section III.D.1) and *Multistage Filters* (Section III.D.2). Their performance is very similar, the main advantage of sample and hold being implementation simplicity, and the main advantage of multistage filters being higher accuracy. In contrast to random sampling, the relative errors of our two new algorithms scale with $1/M$, where M is the amount of SRAM. This allows our algorithms to provide much more accurate estimates than random sampling using the same amount of memory. However, unlike sampled NetFlow, our algorithms access the memory for each packet, so they must use memories fast enough to keep up with line speeds. In Section III.D.3 we present improvements that further increase the accuracy of these algorithms on traces (Section III.H). We start by describing the main ideas behind these schemes.

III.D.1 Sample and hold

Base Idea: The simplest way to identify large flows is through sampling but with the following twist. As with ordinary sampling, we sample each packet with a probability. If a packet is sampled and the flow it belongs to has no entry in the flow memory, a new entry is created. However, after an entry is created for a flow, unlike in sampled NetFlow, we update the entry for **every** subsequent packet belonging to the flow, as shown in Figure III.1. The counting samples of Gibbons and Matias [GM98] use the same core idea.

Thus once a flow is *sampled*, a corresponding counter is *held* in a hash table in flow memory till the end of the measurement interval. While this clearly requires processing (looking up the flow entry and updating a counter) for every packet (unlike Sampled NetFlow), we will show that the reduced memory requirements allow the flow memory to be in SRAM instead of DRAM. This in turn allows the per-packet processing to scale with line speeds.

Let p be the probability with which we sample a byte. Thus the sampling

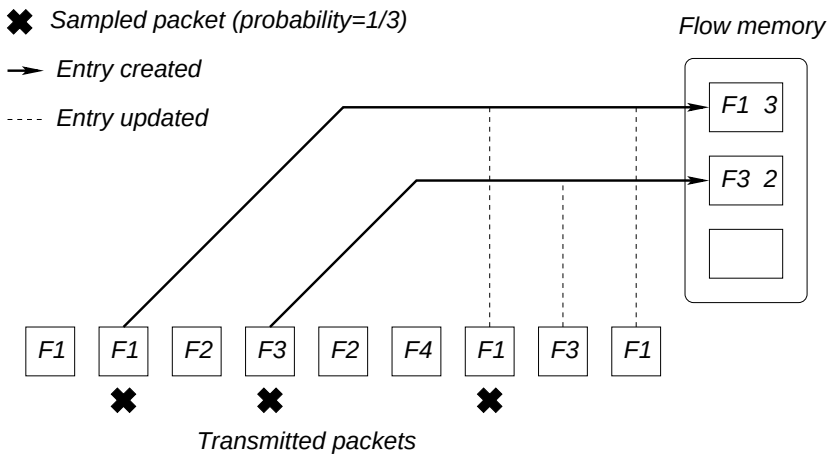


Figure III.1: The leftmost packet with flow label $F1$ arrives first at the router. After an entry is created for a flow (solid line) the counter is updated for all its packets (dotted lines)

probability for a packet of size s is $p_s = 1 - (1-p)^s \approx 1 - e^{-sp}$. This can be looked up in a precomputed table or approximated by $p_s = p * s$ (for example for packets of up to 1500 bytes and $p \leq 10^{-5}$ this approximation introduces errors smaller than 0.76% in p_s). Choosing a high enough value for p guarantees that flows above the threshold are very likely to be detected. Increasing p unduly can cause too many false positives (small flows filling up the flow memory). The advantage of this scheme is that it is easy to implement and yet gives accurate measurements with very high probability.

Preliminary Analysis: The following example illustrates the method and analysis. Suppose we wish to measure the traffic sent by flows that take over 1% of the link capacity in a measurement interval. There are at most 100 such flows. Instead of making our flow memory have just 100 locations, we will allow oversampling by a factor of 100 and keep $10,000$ locations. We wish to sample each byte with probability p such that the average number of samples is $10,000$. Thus if C bytes can be transmitted in the measurement interval, $p = 10,000/C$.

For the error analysis, consider a flow F that takes 1% of the traffic. Thus F sends more than $C/100$ bytes. Since we are randomly sampling each byte with probability $10,000/C$, the probability that F will not be in the flow memory at the end of the

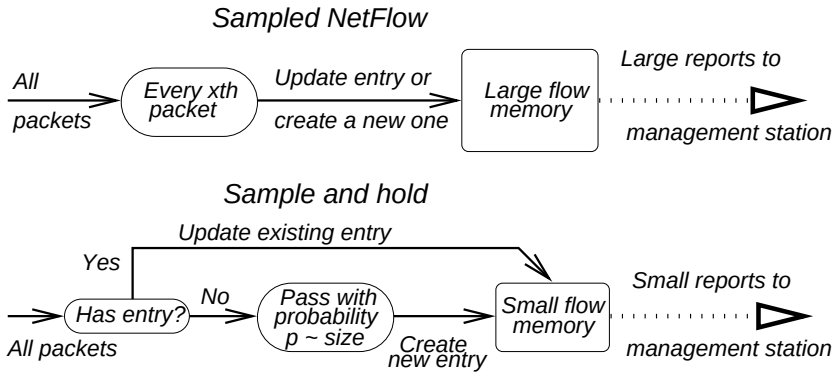


Figure III.2: Sampled NetFlow counts only sampled packets; sample and hold counts all packets after entry created

measurement interval (false negative) is $(1 - 10000/C)^{C/100}$ which is very close to e^{-100}. Notice that the factor of 100 in the exponent is the oversampling factor. Better still, the probability that flow F is in the flow memory after sending 5% of its traffic is, similarly, $1 - e^{-5}$ which is greater than 99% probability. Thus with 99% probability the reported traffic for flow F will be at

most 5% below the actual amount sent by F since once it is in the table we track it precisely.

The analysis can be generalized to arbitrary threshold values; the memory needs to scale inversely with the threshold percentage and directly with the oversampling factor. Notice also that the analysis assumes that there is always space to place a sample flow not already in the memory. Setting $p = 10,000/C$ ensures only that the *average* number of flows sampled[3] is no more than 10,000. However, the distribution of the number of samples is binomial with a small standard deviation (square root of the mean). Thus, adding a few standard deviations to the memory estimate (e.g., a total memory size of 10,300) makes it extremely unlikely that the flow memory will ever overflow[4].

[3]Our analyses from Section III.E.1 andAppendix B.1 also give tight upper bounds on the number of entries used that hold with high probability.

[4]If the flow memory overflows, we can not create new entries until entries are freed at the beginning of the next measurement interval and thus large flows might go undetected. Allocating more memory is probably not an option for hardware implementations. Selectively discarding the least important entries requires us to traverse the entire flow memory and this would violate the strict bounds we have for per packet processing time.

Compared to Sampled NetFlow our idea has three significant differences. Most importantly, we sample only to decide whether to add a flow to the memory; from that point on, we update the flow memory with every byte the flow sends as shown in Figure III.2. As Section III.F shows, this will make our results much more accurate. Second, our sampling technique avoids packet size biases unlike NetFlow which samples every x packets. Third, our technique reduces the extra resource overhead (router processing, router memory, network bandwidth) for sending large reports with many records to a management station.

III.D.2 Multistage filters

Base Idea: The basic multistage filter is shown in Figure III.3. The building blocks are hash stages that operate in parallel. First, consider how the filter operates with only one stage. A stage is a table of counters which is indexed by a hash function computed on a packet flow ID; all counters in the table are initialized to 0 at the start of a measurement interval. When a packet comes in, a hash on its flow ID is computed and the size of the packet is added to the corresponding counter. Since all packets belonging to the same flow hash to the same counter, if a flow F sends more than threshold T, F's counter will exceed the threshold. If we add to the flow memory all packets that hash to counters of T or more, we are guaranteed to identify all the large flows (no false negatives). The multi-stage algorithm of Fang et al [FSGMU98] is similar to our multistage filters and the accounting bins of stochastic fair blue [FKSS01] use a similar data structure to compute drop probabilities for active queue management.

Unfortunately, since the number of counters we can afford is significantly smaller than the number of flows, many flows will map to the same counter. This can cause false positives in two ways: first, small flows can map to counters that hold large flows and get added to flow memory; second, several small flows can hash to the same counter and add up to a number larger than the threshold.

To reduce this large number of false positives, we use multiple stages. Each stage (Figure III.3) uses an *independent* hash function. Only the packets that map to counters of T or more at *all* stages get added to the flow memory. For example, in Figure III.3, if a packet with a flow ID F arrives that hashes to counters 3, 3, and 7 respectively at

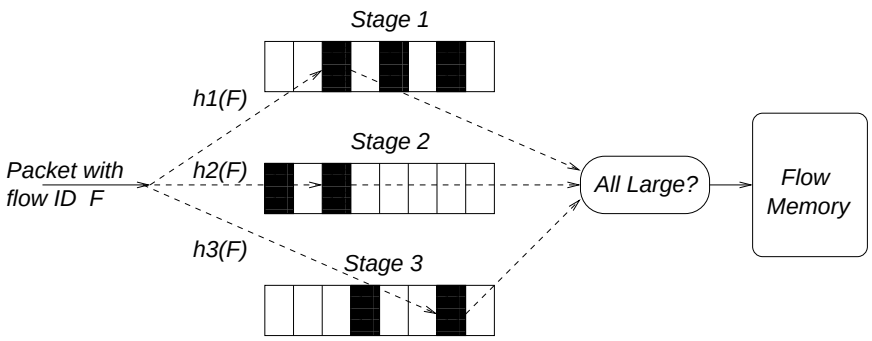


Figure III.3: In a parallel multistage filter, a packet with a flow ID F is hashed using hash function $h1$ into a Stage 1 table, $h2$ into a Stage 2 table, etc. Each table entry contains a counter that is incremented by the packet size. If *all* the hashed counters are above the threshold (shown bolded), F is passed to the flow memory for individual observation.

the three stages, F will pass the filter (counters that are over the threshold are shown darkened). On the other hand, a flow G that hashes to counters 7, 5, and 4 will not pass the filter because the second stage counter is not over the threshold. Effectively, the multiple stages attenuate the probability of false positives exponentially in the number of stages. This is shown by the following simple analysis.

Preliminary Analysis: Assume a 100 Mbytes/s link[5], with 100,000 flows and we want to identify the flows above 1% of the link during a one second measurement interval. Assume each stage has 1,000 buckets and a threshold of 1 Mbyte. Let's see what the probability is for a flow sending 100 Kbytes to pass the filter. For this flow to pass one stage, the other flows need to add up to 1 Mbyte - 100 Kbytes = 900 Kbytes. There are at most 99,900/900=111 such buckets out of the 1,000 at each stage. Therefore, the probability of passing one stage is at most 11.1%. With 4 independent stages, the probability that a certain flow no larger than 100 Kbytes passes all 4 stages is the *product* of the individual stage probabilities which is at most $1.52 * 10^{-4}$.

Based on this analysis, we can dimension the flow memory so that it is large enough to accommodate all flows that pass the filter. The expected number of flows

[5]To simplify computation, in our examples we assume that 1Mbyte=1,000,000 bytes and 1Kbyte=1,000 bytes.

below 100 Kbytes passing the filter is at most $100,000 * 15.2 * 10^{-4} < 16$. There can be at most 999 flows above 100 Kbytes, so the number of entries we expect to accommodate all flows is at most 1,015. Section III.E has a rigorous theorem that proves a stronger bound on the number of flows expected to pass the filter (for this example, 122 entries) that holds for any distribution of flow sizes. Note the potential scalability of the scheme. If the number of flows increases to 1 million, we simply add a fifth hash stage to get the same effect. Thus to handle 100,000 flows requires roughly 4000 counters and a flow memory of approximately 100 memory locations, while to handle 1 million flows requires roughly 5000 counters and the same size of flow memory. This is logarithmic scaling.

The number of memory accesses per packet for a multistage filter is one read and one write per stage. If the number of stages is small, this is feasible even at high speeds by doing parallel memory accesses to each stage in a chip implementation.[6] Multistage filters also need to compute the hash functions. These can be computed efficiently in hardware. For software implementations this adds to the per packet processing and can replace memory accesses as the main bottleneck. However, we already need to compute a hash function to locate the per-flow entries in the flow memory, thus one can argue that we do not introduce a new problem, just make an existing one worse. While multistage filters are more complex than sample-and-hold, they have two important advantages. They reduce the probability of false negatives to 0 and decrease the probability of false positives, thereby reducing the size of the required flow memory.

The serial multistage filter

We briefly present a variant of the multistage filter called a serial multistage filter (Figure III.4). Instead of using multiple stages in parallel, we can place them serially after each other, each stage seeing only the packets that passed the previous stage.

Let d be the number of stages (the depth of the serial filter). We set a *stage threshold* of T/d for all the stages. Thus for a flow that sends T bytes, by the time the last packet is sent, the counters the flow hashes to at all d stages reach T/d, so the packet will pass to the flow memory. As with parallel filters, we have no false negatives. As

[6] We describe details of a preliminary OC-192 chip implementation of multistage filters in Section III.I.

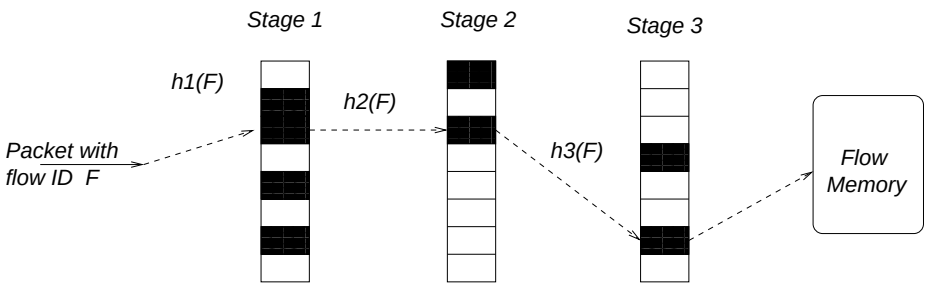


Figure III.4: In a serial multistage filter, a packet with a flow ID F is hashed using hash function $h1$ into a Stage 1 table. If the counter is below the stage threshold T/d, it is incremented. If the counter reaches the stage threshold the packet is hashed using function $h2$ to a Stage 2 counter, etc. If the packet passes all stages, an entry is created for F in the flow memory.

with parallel filters, small flows can pass the filter only if they keep hashing to counters made large by other flows.

The analytical evaluation of serial filters is more complicated than for parallel filters. On one hand the early stages shield later stages from much of the traffic, and this contributes to stronger filtering. On the other hand the threshold used by stages is smaller (by a factor of d) and this contributes to weaker filtering. Since, as shown in Section III.H, parallel filters perform better than serial filters on traces of actual traffic, the main focus in this chapter will be on parallel filters.

III.D.3 Improvements to the basic algorithms

The improvements to our algorithms presented in this section further increase the accuracy of the measurements and reduce the memory requirements. Some of the improvements apply to both algorithms, some apply only to one of them.

Basic optimizations

There are a number of basic optimizations that exploit the fact that large flows often last for more than one measurement interval.

Preserving entries: Erasing the flow memory after each interval implies that the bytes of a large flow sent before the flow is allocated an entry are not counted. By

preserving entries of large flows across measurement intervals and only reinitializing stage counters, *all long lived large flows are measured nearly exactly.* To distinguish between a large flow that was identified late and a small flow that was identified by error, a conservative solution is to preserve the entries of not only the flows for which we count at least T bytes in the current interval, but also all the flows that were added in the current interval (since they may be large flows that entered late).

Early removal: (refinement of preserving entries) Sample and hold has a larger rate of false positives than multistage filters. If we keep for one more interval all the flows that obtained a new entry, many small flows will keep their entries for two intervals. We can improve the situation by selectively removing some of the flow entries created in the current interval. The new rule for preserving entries is as follows. We define an early removal threshold R that is less then the threshold T. At the end of the measurement interval, we keep all entries whose counter is at least T and all entries that have been added during the current interval and whose counter is at least R.

Shielding: Consider large, long-lived flows that go through the filter each measurement interval. Each measurement interval, the counters they hash to exceed the threshold. With shielding, traffic belonging to flows that have an entry in flow memory no longer passes through the filter (the counters in the filter are not incremented for packets with an entry), thereby reducing false positives. If we shield the filter from a large flow, many of the counters it hashes to will not reach the threshold after the first interval. This reduces the probability that a random small flow will pass the filter by hashing to counters that are large because of other flows.

Conservative update of counters

We now describe an important optimization for multistage filters that improves performance by an order of magnitude. *Conservative update* reduces the number of false positives of multistage filters by three subtle changes to the rules for updating counters. In essence, we endeavor to increment counters as little as possible (thereby reducing false positives by preventing small flows from passing the filter) while still avoiding false negatives (i.e., we need to ensure that all flows that reach the threshold still pass the filter.)

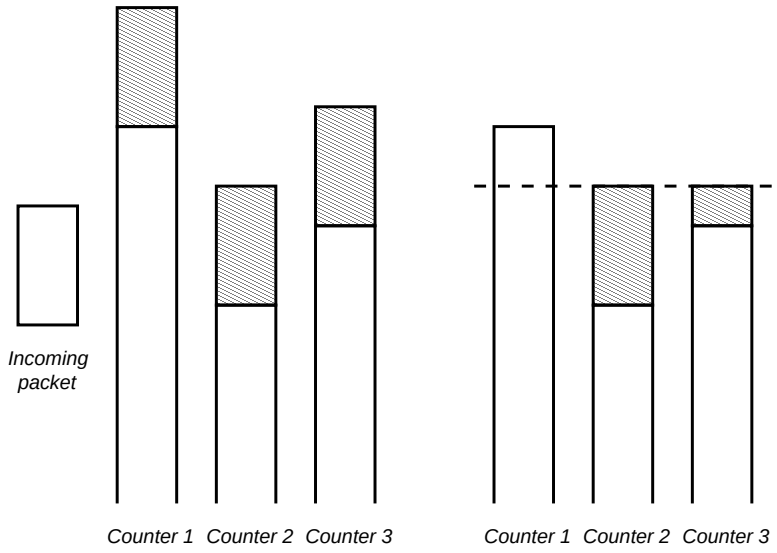


Figure III.5: Conservative update: without conservative update (left), all counters are increased by the size of the incoming packet; with conservative update (right), no counter is increased to more than the size of the smallest counter plus the size of the packet

The first change (Figure III.5) applies only to parallel filters and only for packets that don't pass the filter. As usual, an arriving flow F is hashed to a counter at each stage. We update the smallest of the counters normally (by adding the size of the packet). *However, the other counters are set to the maximum of their old value and the new value of the smallest counter.* Since the amount of traffic sent by the current flow is at most the new value of the smallest counter, this change *cannot introduce a false negative* for the flow the packet belongs to. Since we never decrement counters, other large flows that might hash to the same counters are not prevented from passing the filter.

The second change is very simple and applies to both parallel and serial filters. When a packet passes the filter and it obtains an entry in the flow memory, no counters should be updated. This will leave the counters below the threshold. Other flows with smaller packets that hash to these counters will get less "help" in passing the filter.

The third change applies only to serial filters. It regards the way counters are updated when the threshold is exceeded in any stage but the last one. Let's say the value of the counter a packet hashes to at stage i is $T/d - x$ and the size of the packet is $s > x > 0$. Normally one would increment the counter at stage i to T/d and add $s - x$

to the counter from stage $i+1$. What we can do instead with the counter at stage $i+1$ is update its value to the maximum of $s-x$ and its old value (assuming $s-x < T/d$). Since the counter at stage i was below T/d, we know that no prior packets belonging to the same flow as the current one passed this stage and contributed to the value of the counter at stage $i+1$. We could not apply this change if the threshold T was allowed to change during a measurement interval.

III.E Analytical evaluation of our algorithms

In this section we analytically evaluate our algorithms. We only present the main results. The proofs, supporting lemmas and some of the less important results (e.g., high probability bounds corresponding to our bounds on the average number of flows passing a multistage filter) are in Appendix B.1. We focus on two important questions:

- *How good are the results?* We use two distinct measures of the quality of the results: how many of the large flows are identified, and how accurately is their traffic estimated?
- *What are the resources required by the algorithm?* The key resource measure is the size of flow memory needed. A second resource measure is the number of memory references required.

In Section III.E.1 we analyze our sample and hold algorithm, and in Section III.E.2 we analyze multistage filters. We first analyze the basic algorithms and then examine the effect of some of the improvements presented in Section III.D.3. In the next section (Section III.F) we use the results of this section to analytically compare our algorithms with sampled NetFlow.

Example: We will use the following running example to give numeric instances. Assume a 100 Mbyte/s link with $100,000$ flows. We want to measure all flows whose traffic is more than 1% (1 Mbyte) of link capacity in a one second measurement interval.

III.E.1 Sample and hold

We first define some notation we use in this section.

- p the probability for sampling a byte;
- s the size of a flow (in bytes);
- T the threshold for large flows;
- C the capacity of the link – the number of bytes that can be sent during the *entire* measurement interval;
- O the oversampling factor defined by $p = O \cdot 1/T$;
- c the number of bytes actually counted for a flow.

The quality of results for sample and hold

The first measure of the quality of the results is the probability that a flow at the threshold is not identified. As presented in Section III.D.1, the probability that a flow of size T is not identified is $(1-p)^T \approx e^{-O}$. An oversampling factor of 20 results in a probability of missing flows at the threshold of $2 * 10^{-9}$.

Example: For our example, p must be 1 in 50,000 bytes for an oversampling of 20. With an average packet size of 500 bytes this is roughly 1 in 100 packets.

The second measure of the quality of the results is the difference between the size of a flow s and our estimate. The number of bytes that go by before the first one gets sampled has a geometric probability distribution[7]: it is x with a probability[8] $(1-p)^x p$.

Therefore $E[s-c] = 1/p$ and $SD[s-c] = \sqrt{1-p}/p$. The best estimate for s is $c + 1/p$ and its standard deviation is $\sqrt{1-p}/p$. If we choose to use c as an estimate for s then the error will be larger, but we never overestimate the size of the flow[9]. In this case, the deviation from the actual value of s is $\sqrt{E[(s-c)^2]} = \sqrt{2-p}/p$. Based on this value we can also compute the relative error of a flow of size T which is $T\sqrt{2-p}/p = \sqrt{2-p}/O$.

Example: For our example, with an oversampling factor O of 20, the relative error (computed as the standard deviation of the estimate divided by the actual value) for a flow at the threshold is 7%.

[7] We ignore for simplicity that the bytes before the first sampled byte that are in the same packet with it are also counted. Therefore the actual algorithm will be slightly more accurate than this model.

[8] Since we focus on large flows, we ignore for simplicity the correction factor we need to apply to account for the case when the flow goes undetected (i.e., x is actually bound by the size of the flow s, but we ignore this).

[9] Gibbons and Matias [GM98] have a more elaborate analysis and use a different correction factor.

The memory requirements for sample and hold

The size of the flow memory is determined by the number of flows identified. The actual number of sampled packets is an upper bound on the number of entries needed in the flow memory because new entries are created only for sampled packets. Assuming that the link is constantly busy, by the linearity of expectation, the expected number of sampled bytes is $p \cdot C = O \cdot C/T$.

Example: Using an oversampling of 20 requires 2,000 entries on average.

The number of sampled bytes can exceed this value. Since the number of sampled bytes has a binomial distribution, we can use the normal curve to bound with high probability the number of bytes sampled during the measurement interval. Therefore with probability 99% the actual number will be at most 2.33 standard deviations above the expected value; similarly, with probability 99.9% it will be at most 3.08 standard deviations above the expected value. The standard deviation of the number of sampled bytes is $\sqrt{Cp(1-p)}$ since it has a binomial distribution.

Example: For an oversampling of 20 and an overflow probability of 0.1% we need at most 2,147 entries.

This result can be further tightened if we make assumptions about the distribution of flow sizes and thus account for very large flows having many of their packets sampled. Let's assume that the flows have a Zipf (Pareto) distribution with shape parameter 1 defined as $Prs > x = constant * x^{-1}$. If we have n flows that use the whole bandwidth C, the total traffic of the largest j flows is at least $C\frac{\ln(j+1)}{\ln(2n+1)}$ (see Appendix B.2). For any value of j between 0 and n we obtain an upper bound on the number of entries expected to be used in the flow memory by assuming that the largest j flows always have an entry by having at least one of their packets sampled and each packet sampled from the rest of the traffic creates an entry: $j + Cp(1 - \ln(j+1)/\ln(2n+1)$. By differentiating we obtain the value of j that provides the tightest bound: $j = Cp/\ln(2n+1) - 1$.

Example: Using an oversampling of 20 requires at most 1,328 entries on average.

The effect of preserving entries

We preserve entries across measurement intervals to improve accuracy. The probability of missing a large flow decreases because we cannot miss it if we keep its

entry from the prior interval. Accuracy increases because we know the exact size of the flows whose entries we keep. To quantify these improvements we need to know the ratio of long lived flows among the large ones.

The cost of this improvement in accuracy is an increase in the size of the flow memory. We need enough memory to hold the samples from both measurement intervals[10]. Therefore the expected number of entries is bounded by $2O \cdot C/T$.

To bound with high probability the number of entries we use the normal curve and the standard deviation of the number of sampled packets during the 2 intervals which is $\sqrt{2Cp(1-p)}$.

Example: For an oversampling of 20 and acceptable probability of overflow equal to 0.1%, the flow memory has to have at most 4,207 entries to preserve entries.

The effect of early removal

The effect of early removal on the proportion of false negatives depends on whether or not the entries removed early are reported. Since we believe it is more realistic that implementations will not report these entries, we will use this assumption in our analysis. Let $R < T$ be the early removal threshold. A flow at the threshold is not reported unless one of its first $T - R$ bytes is sampled. Therefore the probability of missing the flow is approximately $e^{-O(T-R)/T}$. If we use an early removal threshold of $R = 0.2 * T$, this increases the probability of missing a large flow from $2 * 10^{-9}$ to $1.1 * 10^{-7}$ with an oversampling of 20.

Early removal reduces the size of the memory required by limiting the number of entries that are preserved from the previous measurement interval. Since there can be at most C/R flows sending R bytes, the number of entries that we keep is at most C/R which can be smaller than OC/T, the bound on the expected number of sampled packets. The expected number of entries we need is $C/R + OC/T$.

To bound with high probability the number of entries we use the normal curve. If $R \geq T/O$, the standard deviation is given only by the randomness of the packets sampled in one interval and is $\sqrt{Cp(1-p)}$.

[10] We actually also keep the older entries that are above the threshold. Since we are performing a worst case analysis, we assume that there is no flow above the threshold, because if there were, many of its packets would be sampled, decreasing the number of entries required.

Example: An oversampling of 20 and $R = 0.2T$ with overflow probability 0.1% requires 2,647 memory entries.

III.E.2 Multistage filters

In this section, we analyze parallel multistage filters. We first define some new notation:

- b the number of buckets in a stage;
- d the depth of the filter (the number of stages);
- n the number of active flows;
- k the "stage strength" is the ratio of the threshold and the average size of a counter. $k = \frac{T\ b}{C}$, where C denotes the channel capacity as before. Intuitively, this is the factor we inflate each stage memory beyond the minimum of C/T .

Example: To illustrate our results numerically, we will assume that we solve the measurement example described in Section III.E with a 4 stage filter, with 1000 buckets at each stage. The stage strength k is 10 because each stage memory has 10 times more buckets than the maximum number of flows (i.e., 100) that can cross the specified threshold of 1%.

The quality of results for multistage filters

As discussed in Section III.D.2, multistage filters have no false negatives. The error of the traffic estimates for large flows is bounded by the threshold T since no flow can send T bytes without being entered into the flow memory. The stronger the filter, the less likely it is that the flow will be entered into the flow memory much before it reaches T. We first state an upper bound for the probability of a small flow passing the filter described in Section III.D.2.

Lemma 1 *Assuming the hash functions used by different stages are independent, the probability of a flow of size $s < T(1 - 1/k)$ passing a parallel multistage filter is at most $p_s \leq \left(\frac{1}{k}\frac{T}{T-s}\right)^d$.*

The proof of this bound presented in Appendix B.1 formalizes the preliminary analysis of multistage filters from Section III.D.2. Note that the bound *makes no assumption about the distribution of flow sizes*, and thus applies for all flow distributions. We only assume that the hash functions are random and independent. The bound is tight in the sense that it is almost exact for a distribution that has $\lfloor (C-s)/(T-s) \rfloor$ flows of size $(T-s)$ that send all their packets before the flow of size s. However, for realistic traffic mixes (e.g., if flow sizes follow a Zipf distribution), this is a very conservative bound.

Based on this lemma we obtain a lower bound for the expected error for a large flow.

Theorem 2 *The expected number of bytes of a large flow of size s undetected by a multistage filter is bound from below by*

$$E[s-c] \geq T\left(1 - \frac{d}{k(d-1)}\right) - y_{max} \tag{III.1}$$

where y_{max} is the maximum size of a packet.

This bound suggests that we can significantly improve the accuracy of the estimates by adding a correction factor to the bytes actually counted. The down side to adding a correction factor is that we can overestimate some flow sizes; this may be a problem for accounting applications. The y_{max} factor from the result comes from the fact that when the packet that makes the counters exceed the threshold arrives, c is initialized to its size which can be as much as y_{max}.

The memory requirements for multistage filters

We can dimension the flow memory based on bounds on the number of flows that pass the filter. Based on Lemma 1 we can compute a bound on the total number of flows expected to pass the filter (the full derivation of this theorem is in Appendix B.1).

Theorem 3 *The expected number of flows passing a parallel multistage filter is bound by*

$$E[n_{pass}] \leq max\left(\frac{b}{k-1}, n\left(\frac{n}{kn-b}\right)^d\right) + n\left(\frac{n}{kn-b}\right)^d \tag{III.2}$$

Example: Theorem 3 gives a bound of 121.2 flows. Using 3 stages would have resulted in a bound of 200.6 and using 5 would give 112.1. Note that when the first term dominates the max, there is not much gain in adding more stages.

We can also bound the number of flows passing the filter with high probability.

Example: The probability that more than 185 flows pass the filter is at most 0.1%. Thus by increasing the flow memory from the expected size of 122 to 185 we can make overflow of the flow memory extremely improbable.

As with sample and hold, making assumptions about the distribution of flow sizes can lead to a smaller bound on the number of flows expected to enter the flow memory.

Theorem 4 *If the flows sizes have a Zipf distribution with parameter 1, the expected number of flows passing a parallel multistage filter is bound by*

$$E[n_{pass}] \leq i_0 + \frac{n}{k^d} + \frac{db}{k^{d+1}} + \frac{db\ln(n+1)^{d-2}}{k^2\left(k\ln(n+1) - \frac{b}{i_0-0.5}\right)^{d-1}} \quad \text{(III.3)}$$

where $i_0 = \lceil max(1.5 + \frac{b}{k\ln(n+1)}, \frac{b}{\ln(2n+1)(k-1)})\rceil$.

Example: Theorem 4 gives a bound of 21.7 on the number of flows expected to pass the filter.

The effect of preserving entries and shielding

Preserving entries affects the accuracy of the results the same way as for sample and hold: long-lived large flows have their traffic counted exactly after their first interval above the threshold. As with sample and hold, preserving entries basically doubles all the bounds for memory usage.

Shielding has a strong effect on filter performance, since it reduces the traffic presented to the filter. Reducing the traffic α times increases the stage strength to $k * \alpha$, which can be substituted in Theorems 2 and 3.

III.F Comparing Measurement Methods

In this section we analytically compare the performance of three traffic measurement algorithms: our two new algorithms (sample and hold and multistage filters)

Measure	Sample and hold	Multistage filters	Sampling
Relative error for a flow of size zC	$\frac{\sqrt{2}}{Mz}$	$\frac{1+10\ r\ log_{10}(n)}{Mz}$	$\frac{1}{\sqrt{Mz}}$
Memory accesses per packet	1	$1 + log_{10}(n)$	$\frac{1}{x} = \frac{M}{C}$

Table III.1: Comparison of the core algorithms: sample and hold provides most accurate results while pure sampling has very few memory accesses

and Sampled NetFlow. First, in Section III.F.1, we compare the algorithms at the core of traffic measurement devices. For the core comparison, we assume that each of the algorithms is given the *same* amount of high speed memory and we compare their accuracy and number of memory accesses. This allows a fundamental analytical comparison of the effectiveness of each algorithm in identifying heavy-hitters.

However, in practice, it may be unfair to compare Sampled NetFlow with our algorithms using the same amount of memory. This is because Sampled NetFlow can afford to use a large amount of DRAM (because it does not process every packet) while our algorithms cannot (because they process every packet and hence need to store per flow entries in SRAM). Thus we perform a second comparison in Section III.F.2 of complete traffic measurement devices. In this second comparison, we allow Sampled NetFlow to use more memory than our algorithms. The comparisons are based on the algorithm analysis in Section III.E and an analysis of NetFlow presented in Appendix A.

III.F.1 Comparison of the core algorithms

In this section we compare sample and hold, multistage filters and ordinary sampling (used by NetFlow) under the assumption that they are all constrained to using M memory entries. More precisely, the expected number of memory entries used is at most M irrespective of the distribution of flow sizes. We focus on the accuracy of the measurement of a flow (defined as the standard deviation of an estimate over the actual size of the flow) whose traffic is zC (for flows of 1% of the link capacity we would use $z = 0.01$).

The bound on the expected number of entries is the same for sample and hold and for sampling and is pC. By making this equal to M we can solve for p.

By substituting in the formulae we have for the accuracy of the estimates and after eliminating some terms that become insignificant (as p decreases and as the link capacity goes up) we obtain the results shown in Table III.1.

For multistage filters, we use a simplified version of the result from Theorem 3: $E[n_{pass}] \leq b/k + n/k^d$. We increase the number of stages used by the multistage filter logarithmically as the number of flows increases so that only a single small flow is expected to pass the filter[11] and the strength of the stages is 10. At this point we estimate the memory usage to be $M = b/k + 1 + rbd = C/T + 1 + r10log_{10}(n)C/T$, where $r < 1$ depends on the implementation and reflects the relative cost of a counter and an entry in the flow memory. From here we obtain T, which will be an upper bound on the error of our estimate of flows of size zC. The result from Table III.1 is immediate.

The term Mz that appears in all formulae in the first row of the table is exactly equal to the oversampling we defined in the case of sample and hold. It expresses how many times we are willing to allocate over the theoretical minimum memory to obtain better accuracy. We can see that the error of our algorithms decreases inversely proportional to this term while the error of sampling is proportional to the inverse of its square root.

The second line of Table III.1 gives the number of memory locations accessed per packet by each algorithm. Since sample and hold performs a packet lookup for every packet[12], its per packet processing is 1. Multistage filters add to the one flow memory lookup an extra access to one counter per stage and the number of stages increases as the logarithm of the number of flows. Finally, for ordinary sampling, one in $x = C/M$ packets get sampled so the average per packet processing is $1/x = M/C$.

Table III.1 provides a fundamental comparison of our new algorithms with ordinary sampling as used in Sampled NetFlow. The first line shows that the relative error of our algorithms scales with $1/M$ which is much better than the $1/\sqrt{M}$ scaling of ordinary sampling. However, the second line shows that this improvement comes at the

[11] Configuring the filter such that a small number of small flows pass would have resulted in smaller memory and fewer memory accesses (because we would need fewer stages), but it would have complicated the formulae.

[12] We equate a lookup in the flow memory to a single memory access. This is true if we use a content addressable memory. Lookups without hardware support require a few more memory accesses to resolve hash collisions.

cost of requiring at least one memory access per packet for our algorithms. While this allows us to implement the new algorithms using SRAM, the smaller number of memory accesses ($\ll 1$) per packet allows Sampled NetFlow to use DRAM. This is true as long as x is larger than the ratio of a DRAM memory access to an SRAM memory access. However, even a DRAM implementation of Sampled NetFlow has some problems which we turn to in our second comparison.

III.F.2 Comparing Measurement Devices

Table III.1 implies that increasing DRAM memory size M to infinity can reduce the relative error of Sampled NetFlow to zero. But this assumes that by increasing memory one can increase the sampling rate so that x becomes arbitrarily close to 1. If $x = 1$, there would be no error since every packet is logged. But x must at least be as large as the ratio of DRAM speed (currently around 60 ns) to SRAM speed (currently around 5 ns); thus Sampled NetFlow will always have a minimum error corresponding to this value of x even when given unlimited DRAM.

With this insight, we now compare the performance of our algorithms and NetFlow in Table III.2 without limiting NetFlow memory. Thus Table III.2 takes into account the underlying technologies (i.e., the potential use of DRAM over SRAM) and one optimization (i.e., preserving entries) for both of our algorithms.

We consider the task of estimating the size of all the flows above a fraction z of the link capacity over a measurement interval of t seconds. In order to make the comparison possible we change somewhat the way NetFlow operates: we assume that it reports the traffic data for each flow after each measurement interval, like our algorithms do. The four characteristics of the traffic measurement algorithms presented in the table are: the percentage of large flows known to be measured exactly, the relative error of the estimate of a large flow, the upper bound on the memory size and the number of memory accesses per packet.

Note that the table does not contain the actual memory used but a bound. For example the number of entries used by NetFlow is bounded by the number of active flows and the number of DRAM memory lookups that it can perform during a measure-

ment interval (which doesn't change as the link capacity grows)[13]. Our measurements in Section III.H show that for all three algorithms the actual memory usage is much smaller than the bounds, especially for multistage filters. Memory is measured in entries, not bytes. We assume that a flow memory entry is equivalent to 10 of the counters used by the filter (i.e. $r = 1/10$) because the flow ID is typically much larger than the counter. Note that the number of memory accesses required per packet does not necessarily translate to the time spent on the packet because memory accesses can be pipelined or performed in parallel.

We make simplifying assumptions about technology evolution. As link speeds increase, so must the electronics[14]. Therefore we assume that SRAM speeds keep pace with link capacities. We also assume that the speed of DRAM does not improve significantly ([PH98] states that DRAM speeds improve only at 9% per year while clock rates improve at 40% per year).

We assume the following configurations for the three algorithms. Our algorithms preserve entries. O is the oversampling used by sample and hold. For multistage filters we introduce a new parameter expressing how many times larger a flow of interest is than the threshold of the filter $u = zC/T$. Since the speed gap between the DRAM used by sampled NetFlow and the link speeds increases as link speeds increase, NetFlow has to decrease its sampling rate proportionally with the increase in capacity[15] to provide the smallest possible error. For the NetFlow error calculations we also assume that the size of the packets of large flows is 1500 bytes.

Besides the differences that stem from the core algorithms (Table III.1), we see new differences in Table III.2. The first big difference (Row 1 of Table III.2) is that unlike NetFlow, *our algorithms provide exact measures for long-lived large flows* by preserving entries. More precisely, by preserving entries our algorithms will exactly measure traffic for all (or almost all in the case of sample and hold) of the large flows that were large in the previous interval. Given that our measurements show that most large flows are long

[13]The limit on the number of packets NetFlow can process we used for Table III.2 is based on Cisco documentation that states that sampling should be turned on for speeds larger than OC-3 (155.52 Mbits/second). Thus we assumed that this is the maximum speed at which NetFlow can handle minimum sized (40 byte) packets.

[14]All optical routers are not a viable solution today because buffers cannot be implemented with optics.

[15]If the capacity of the link is x times OC-3, then one in x packets gets sampled. We assume based on [CN] that NetFlow can handle packets no smaller than 40 bytes at OC-3 speeds.

Measure	Sample and hold	Multistage filters	Sampled NetFlow
Exact measurements	$\lessapprox longlived\%$	$longlived\%$	0
Relative error	$1.41/O$	$\lessapprox 1/u$	$0.0088/\sqrt{zt}$
Memory bound	$2O/z$	$2/z + 1/z\ log_{10}(n)$	min(n,486000 t)
Memory accesses	1	$1 + log_{10}(n)$	$1/x$

Table III.2: Comparison of traffic measurement devices

lived (depending on the flow definition, the average percentage of the large flows that were large in the previous measurement interval is between 56% and 81%), this is a big advantage.

Of course, one could get the same advantage by using an SRAM flow memory that preserves large flows across measurement intervals in Sampled NetFlow as well. However, that would require the router to root through its DRAM flow memory before the end of the interval to find the large flows, a large processing load. One can also argue that if one can afford an SRAM flow memory, it is quite easy to do sample and hold.

The second big difference (Row 2 of Table III.2) is that we can make our algorithms arbitrarily accurate at the cost of increases in the amount of memory used[16] while sampled NetFlow can do so only by increasing the measurement interval t.

The third row of Table III.2 compares the memory used by the algorithms. The extra factor of 2 for sample and hold and multistage filters arises from preserving entries. Note that the number of entries used by Sampled NetFlow is bounded by both the number n of active flows and the number of memory accesses that can be made in t seconds. Finally, the fourth row of Table III.2 is identical to the second row of Table III.1.

Table III.2 demonstrates that our algorithms have two advantages over NetFlow: **i)** they provide exact values for long-lived large flows (row 1) and **ii)** they provide much better accuracy even for small measurement intervals (row 2). Besides these advantages, our algorithms also have three more advantages not shown in Table III.2. These are **iii)** our estimates are provable lower bounds on traffic, **iv)** reduced resource consumption for collection, and **v)** faster detection of new large flows. We now examine

[16] Of course, technology and cost impose limitations on the amount of available SRAM but the current limits for on and off-chip SRAM are high enough for our algorithms.

advantages **iii)** and **iv)** in more detail.

iii) Provable Lower Bounds: A possible disadvantage of Sampled NetFlow is that the NetFlow estimate is not an actual lower bound on the flow size. Thus a customer may be charged for more than the customer sends. While one can make the probability of overcharging arbitrarily low (using large measurement intervals or other methods from [DLT01]), there may be philosophical objections to overcharging. Our algorithms do not have this problem.

iv) Reduced Resource Consumption: Clearly, while Sampled NetFlow can increase DRAM to improve accuracy, the router has more entries at the end of the measurement interval. These records have to be processed, potentially aggregated, and transmitted over the network to the management station. If the router extracts the heavy hitters from the log, then router processing is large; if not, the bandwidth consumed and processing at the management station is large. By using fewer entries, our algorithms avoid these resource (e.g., memory, transmission bandwidth, and router CPU cycles) bottlenecks, but as detailed in Table III.2 sample and hold and multistage filters incur more upfront work by processing each packet.

III.G Dimensioning traffic measurement devices

We describe how to dimension our algorithms. For applications that face adversarial behavior (e.g., detecting DoS attacks), one should use the conservative bounds from Sections III.E.1 and III.E.2. Other applications such as accounting can obtain better accuracy from more aggressive dimensioning as described below. The measurements from Section III.H show that the gains can be substantial. For example the number of false positives for a multistage filter can be four orders of magnitude below what the conservative analysis predicts. To avoid relying on a priori knowledge of flow distributions, we adapt algorithm parameters to actual traffic. The main idea is to *keep decreasing the threshold below the conservative estimate until the flow memory is nearly full* (totally filling the memory can result in new large flows not being tracked). We defer the discussion of the details of threshold adaptation until Chapter VI.

III.G.1 Dimensioning the multistage filter

Even if we have the correct constants for the threshold adaptation algorithm, there are other configuration parameters for the multistage filter we need to set. Our aim in this section is not to derive the exact optimal values for the configuration parameters of the multistage filters. Due to the dynamic threshold adaptation, the device will work even if we use suboptimal values for the configuration parameters. Nevertheless we want to avoid using configuration parameters that would lead the dynamic adaptation to stabilize at a value of the threshold that is significantly higher than the one for the optimal configuration.

We assume that design constraints limit the total amount of memory we can use for the stage counters and the flow memory, but we have no restrictions on how to divide it between the filter and the flow memory. Since the number of per packet memory accesses might be limited, we assume that we might have a limit on the number of stages. We want to see how we should divide the available memory between the filter and the flow memory and how many stages to use. We base our configuration parameters on some knowledge of the traffic mix (the number of active flows and the percentage of large flows that are long lived).

We first introduce a simplified model of how the multistage filter works. Measurements confirm this model is closer to the actual behavior of the filters than the conservative analysis. Because of shielding, the old large flows do not affect the filter. We assume that because of conservative update only the counters to which the new large flows hash reach the threshold. Let l be the number of large flows and Δl be the number of new large flows. We approximate the probability of a small flow passing one stage by $\Delta l/b$ and of passing the whole filter by $(\Delta l/b)^d$. This gives us the number of false positives in each interval $fp = n(\Delta l/b)^d$. The number of memory locations used at the end of a measurement interval consists of the large flows and the false positives of the previous interval and the new large flows and the new false positives $m = l + \Delta l + 2 * fp$. To be able to establish a tradeoff between using the available memory for the filter or the flow memory, we need to know the relative cost of a counter and a flow entry. Let r denote the ratio between the size of a counter and the size of an entry. The amount of memory used by the filter is going to be equivalent to $b * d * r$ entries. To determine

the optimal number of counters per stage given a certain number of large flows, new large flows and stages, we take the derivative of the total memory with respect to b. Equation III.4 gives the optimal value for b and Equation III.5 gives the total amount of memory required with this choice of b.

$$b = \Delta l \left(\sqrt[d+1]{\frac{2n}{r\Delta l}} \right) \tag{III.4}$$

$$m_{total} = l + \Delta l + (d+1)r\Delta l \left(\sqrt[d+1]{\frac{2n}{r\Delta l}} \right) \tag{III.5}$$

We make a further simplifying assumption that the ratio between Δl and l (related to the flow arrival rate) doesn't depend on the threshold. Our measurements confirm that this is a good approximation for wide ranges of the threshold. For the MAG trace (see Section III.H for a description of the traces we use), when we define the flows at the granularity of TCP connections $\Delta l/l$ is around 44%, when defining flows based on destination IP 37% and when defining them as AS pairs 19%. Let M be the number of entries the available memory can hold. We solve Equation III.5 with respect to l for all possible values of d from 2 to the limit on the number of memory accesses we can afford per packet. We choose the depth of the filter that gives the largest l and compute b based on that value.

III.H Measurements

In Section III.E and Section III.F we used *theoretical* analysis to understand the effectiveness of our algorithms. In this section, we turn to *experimental* analysis to show that our algorithms behave much better on real traces than the (reasonably good) bounds provided by the earlier theoretical analysis and compare them with Sampled NetFlow.

We start by describing the traces we use and some of the configuration details common to all our experiments. In Section III.H.1 we compare the measured performance of the sample and hold algorithm with the predictions of the analytical evaluation, and also evaluate how much the various improvements to the basic algorithm help. In Section III.H.1 we evaluate the multistage filter and the improvements that apply to

Trace	Number of flows (min/avg/max)			MB/5 sec.
	5-tuple	destination IP	AS pair	(min/max)
MAG+	93,437/98,424/105,814	40,796/42,915/45,299	7,177/7,401/7,775	201.0/284.2
MAG	99,264/100,105/101,038	43,172/43,575/43,987	7,353/7,408/7,477	255.8/273.5
IND	13,746/14,349/14,936	8,723/8,933/9,081	-	91.37/99.70
COS	5,157/5,497/5,784	1,124/1,146/1,169	-	14.28/18.70

Table III.3: The traces used for our measurements

it. We conclude with Section III.H.2 where we compare complete traffic measurement devices using our two algorithms with Cisco's Sampled NetFlow.

We use 3 unidirectional traces of Internet traffic: a 4,515 second "clear" one (MAG+) from CAIDA (captured in August 2001 on an OC-48 backbone link between two ISPs) and two 90 second anonymized traces from the MOAT project of NLANR (captured in September 2001 at the access points to the Internet of two large universities on an OC-12 (IND) and an OC-3 (COS)). For some of the experiments use only the first 90 seconds of trace MAG+ as trace MAG.

In our experiments we use 3 different definitions for flows. The first definition is at the granularity of TCP connections: flows are defined by the 5-tuple of source and destination IP address and port and the protocol number. This definition is close to that of Cisco NetFlow. The second definition uses the destination IP address as a flow identifier. This is a definition one could use to identify at a router ongoing (distributed) denial of service attacks. The third definition uses the source and destination autonomous system as the flow identifier. This is close to what one would use to determine traffic patterns in the network. We cannot use this definition with the anonymized traces (IND and COS) because we cannot perform route lookups on them.

Table III.3 describes the traces we used. The number of active flows is given for all applicable flow definitions. The reported values are the smallest, largest and average value over the measurement intervals of the respective traces. The number of megabytes per interval is also given as the smallest and largest value. Our traces use only between 13% and 27% of their respective link capacities[17].

The best value for the size of the measurement interval depends both on the

[17]Table III.3 measures the traffic in megabytes and the link speeds are in multiples of bits per second.

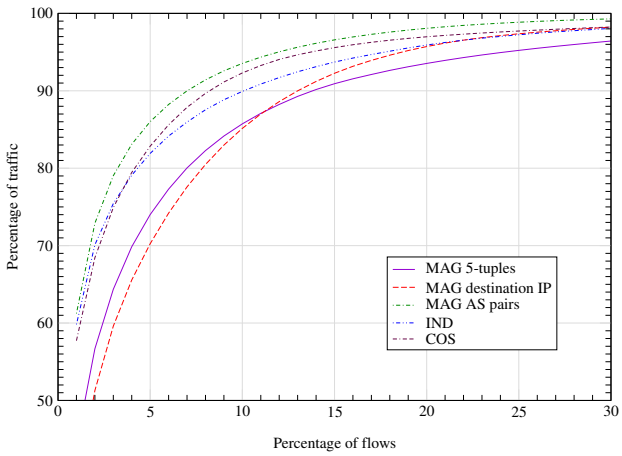


Figure III.6: Cumulative distribution of flow sizes for various traces and flow definitions

application and the traffic mix. We chose to use a measurement interval of 5 seconds in all our experiments. [EV02] gives the measurements we base this decision on. Here we only note that in all cases, across all flows 99% or more of the packets (weighted by packet size) arrive within 5 seconds of the previous packet belonging to the same flow.

Since our algorithms are based on the assumption that a few heavy flows dominate the traffic mix, we find it useful to see to what extent this is true for our traces. Figure III.6 presents the cumulative distributions of flow sizes for the traces MAG, IND and COS for flows defined by 5-tuples. For the trace MAG we also plot the distribution for the case where flows are defined based on destination IP address, and for the case where flows are defined based on the source and destination ASes. As we can see, the top 10% of the flows represent between 85.1% and 93.5% of the total traffic, validating our original assumption that a few flows dominate.

III.H.1 Comparing Theory and Practice

Here we summarize our most important results that compare the theoretical bounds with the results on actual traces, and quantify the benefits of various optimizations. In Appendix D.1 we discuss more measurement results for sample and hold and

Algorithm	Maximum memory usage (entries)/ Average error				
	MAG 5tuple	MAG dstIP	MAG ASpair	IND 5tuple	COS 5tuple
General bound	16,385/25%	16,385/25%	16,385/25%	16,385/25%	16,385/25%
Zipf bound	8,148/25%	7,441/25%	5,489/25%	6,303/25%	5,081/25%
Sample and hold	2,303/24.3%	1,964/24.1%	714/24.40%	1,313/23.8%	710/22.17%
+pres. entries	3,832/4.67%	3,213/3.28%	1,038/1.32%	1,894/3.04%	1,017/6.6%
+early removal	2,659/3.89%	2,294/3.16%	803/1.18%	1,525/2.92%	859/5.46%

Table III.4: Summary of sample and hold measurements for a threshold of 0.025% and an oversampling of 4

in Appendix D.2 more results for multistage filters.

Summary of findings about sample and hold

Table III.4 summarizes our results for a single configuration: a threshold of 0.025% of the link with an oversampling of 4. We ran 50 experiments (with different random hash functions) on each of the reported traces with the respective flow definitions. The table gives the maximum memory usage over the 900 measurement intervals and the ratio between average error for large flows and the threshold.

The first row presents the *theoretical* bounds that hold without making any assumption about the distribution of flow sizes and the number of flows. These are not the bounds on the expected number of entries used (which would be 16,000 in this case), but high probability bounds on the number of entries.

The second row presents *theoretical* bounds assuming that we know the number of flows and know that their sizes have a *Zipf distribution* with a parameter of $\alpha = 1$. Note that the relative errors predicted by theory may appear large (25%) but these are computed for a very low threshold of 0.025% and only apply to flows exactly at the threshold.[18]

The third row shows the actual values we measured for the basic sample and hold algorithm. The actual memory usage is much below the bounds. The first reason is that the links are lightly loaded and the second reason (partially captured by the

[18] We defined the relative error by dividing the average error by the size of the threshold. We could have defined it by taking the average of the ratio of a flow's error to its size but this makes it difficult to compare results from different traces.

analysis that assumes a Zipf distribution of flows sizes) is that large flows have many of their packets sampled. The average error is very close to its expected value.

The fourth row presents the effects of preserving entries. While this increases memory usage (especially where large flows do not have a big share of the traffic) it significantly reduces the error for the estimates of the large flows, because there is no error for large flows identified in previous intervals. This improvement is most noticeable when we have many long-lived flows.

The last row of the table reports the results when preserving entries as well as using an early removal threshold of 15% of the threshold (see Appendix D.1 for why this is a good value). We compensated for the increase in the probability of false negatives early removal causes by increasing the oversampling to 4.7. The average error decreases slightly. The memory usage decreases, especially in the cases where preserving entries caused it to increase most.

We performed measurements on many more configurations. The results are in general similar to the ones from Table III.4, so we only emphasize some noteworthy differences. First, when the expected error approaches the size of a packet, we see significant decreases in the average error. Our analysis assumes that we sample at the byte level. In practice, if a certain packet gets sampled all its bytes are counted, including the ones before the byte that was sampled.

Second, preserving entries reduces the average error by 70% - 95% and increases memory usage by 40% - 70%. These figures do not vary much as we change the threshold or the oversampling. Third, an early removal threshold of 15% reduces the memory usage by 20% - 30%. The size of the improvement depends on the trace and flow definition and it increases slightly with the oversampling.

Summary of findings about multistage filters

Figure III.7 summarizes our findings about configurations with a stage strength of $k = 3$ for our most challenging trace: MAG with flows defined at the granularity of TCP connections. It represents the percentage of small flows (log scale) that passed the filter for depths from 1 to 4 stages. We used a threshold of a 4096th of the maximum traffic. The first (i.e., topmost and solid) line represents the bound of Theorem 3. The

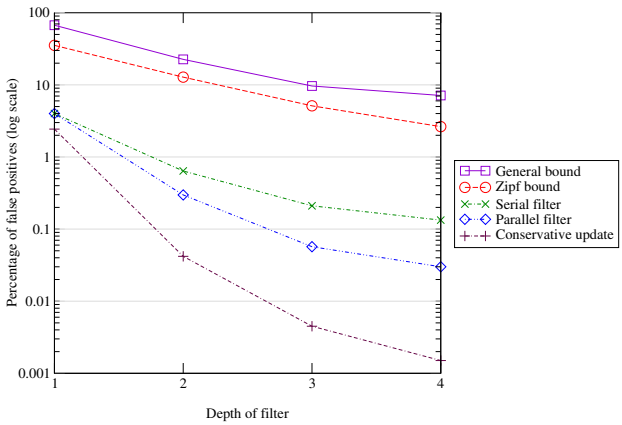


Figure III.7: Filter performance for a stage strength of k=3

second line below represents the improvement in the theoretical bound when we assume a Zipf distribution of flow sizes. Unlike in the case of sample and hold, we always used the maximum traffic, not the link capacity for computing the theoretical bounds. This results in much tighter theoretical bounds.

The third line represents the measured average percentage of false positives of a serial filter, while the fourth line represents a parallel filter. We can see that both are at least 10 times better than the stronger of the theoretical bounds. As the number of stages goes up, the parallel filter gets better than the serial filter by up to a factor of 4. The last line represents a parallel filter with conservative update, which gets progressively better than the parallel filter by up to a factor of 20 as the number of stages increases. We can see that all lines are only slightly concave; this indicates that the percentage of false positives decreases almost exponentially with the number of stages.

Measurements on other traces show similar results. The difference between the bounds and measured performance is even larger for the traces where the largest flows are responsible for a large share of the traffic. With conservative update and without preserving entries the average error is very close to the threshold. Preserving entries reduces the average error in the estimates by 70% to 85% because the estimates for long lived flows are exact in all but the first measurement interval. The improvements in the

results due to preserving entries depend on the traffic mix. Preserving entries increases the number of flow memory entries used by up to 30%. By effectively increasing stage strength k, shielding considerably strengthens weak filters. This can lead to reducing the number of entries by as much as 70%.

III.H.2 Evaluation of complete traffic measurement devices

We now present our final comparison between sample and hold, multistage filters and sampled NetFlow. We perform the evaluation on our long OC-48 trace, MAG+. We assume that our devices can use 1 Mbit of memory (4096 entries[19]), which is well within the possibilities of today's chips. Sampled NetFlow is given unlimited memory and uses a sampling of 1 in 16 packets. We run each algorithm 16 times on the trace with different sampling or hash functions.

Both our algorithms use the adapt the threshold dynamically. To avoid the effect of initial misconfiguration, we ignore the first 10 intervals to give the devices time to reach a relatively stable value for the threshold. We impose a limit of 4 stages for the multistage filters. Based on heuristics presented in Section III.G.1, we use 3114 counters[20] for each stage and 2539 entries of flow memory when using a flow definition at the granularity of TCP connections, 2646 counters and 2773 entries when using the destination IP as flow identifier and 1502 counters and 3345 entries when using the source and destination AS. Multistage filters use shielding and conservative update. Sample and hold uses an oversampling of 4 and an early removal threshold of 15%.

Our purpose is to see how accurately the algorithms measure the largest flows, but there is no implicit definition of what large flows are. We look separately at how well the devices perform for three reference groups: very large flows (above one thousandth of the link capacity), large flows (between one thousandth and a tenth of a thousandth) and medium flows (between a tenth of a thousandth and a hundredth of a thousandth – 15,552 bytes).

For each of these groups we look at two measures of accuracy that we average over all runs and measurement intervals: the percentage of flows not identified and the

[19] Cisco NetFlow uses 64 bytes per entry in cheap DRAM. We assume that the size of a flow memory entry will be 32 bytes (even though 16 or 24 are also plausible).

[20] We conservatively assume that we use 4 bytes for a counter even though 3 bytes would be enough.

relative average error. We compute the relative average error by dividing the sum of the absolute values of all errors by the sum of the sizes of all flows. We use the modulus so that positive and negative errors don't cancel out for NetFlow. For the unidentified flows, we consider that the error is equal to their total traffic. Tables III.5 to III.7 present the results for the 3 different flow definitions.

When using the source and destination AS as flow identifier, the situation is different from the other two cases because the average number of active flows (7,401) is not much larger than the number of memory locations that we can accommodate in our SRAM (4,096), so we will discuss this case separately. In the first two cases, we can see that both our algorithms are much more accurate than sampled NetFlow for large and very large flows. For medium flows the average error is roughly the same, but our algorithms miss more of them than sampled NetFlow. Since sample and hold stabilized at thresholds slightly above 0.01% and multistage filters around 0.002% it is normal that so many of the flows from the third group are not detected.

We believe these results (and similar results not presented here) confirm that our algorithms are better than sampled NetFlow at measuring large flows. Multistage filters are always slightly better than sample and hold despite the fact that we have to sacrifice part of the memory for stage counters. However, tighter algorithms for threshold adaptation can possibly improve both algorithms.

In the third case since the average number of very large, large and medium flows (1,107) was much below the number of available memory locations and these flows were mostly long lived, both of our algorithms measured all these flows very accurately. Thus, even when the number of flows is only a few times larger than the number of active flows, our algorithms ensure that the available memory is used to accurately measure the largest of the flows and provide graceful degradation in case that the traffic deviates very much from the expected (e.g. more flows).

III.I Implementation Issues

We briefly describe implementation issues. Sample and Hold is easy to implement even in a network processor because it adds only one memory reference to packet

Group (flow size)	Unidentified flows / Average error		
	Sample and hold	Multistage filters	Sampled NetFlow
$> 0.1\%$	0% / 0.07508%	0% / 0.03745%	0% / 9.020%
$0.1 \ldots 0.01\%$	1.797% / 7.086%	0% / 1.090%	0.02132% / 22.02%
$0.01 \ldots 0.001\%$	77.01% / 61.20%	54.70% / 43.87%	17.72% / 50.27%

Table III.5: Comparison of traffic measurement devices with flow IDs defined by 5-tuple

Group (flow size)	Unidentified flows / Average error		
	Sample and hold	Multistage filters	Sampled NetFlow
$> 0.1\%$	0% / 0.02508%	0% / 0.01430%	0% / 5.720%
$0.1 \ldots 0.01\%$	0.4289% / 3.153%	0% / 0.9488%	0.01381% / 20.77%
$0.01 \ldots 0.001\%$	65.72% / 51.19%	49.91% / 39.91%	11.54% / 46.59%

Table III.6: Comparison of traffic measurement devices with flow IDs defined by destination IP

Group (flow size)	Unidentified flows / Average error		
	Sample and hold	Multistage filters	Sampled NetFlow
$> 0.1\%$	0% / 0.000008%	0% / 0.000007%	0% / 4.877%
$0.1 \ldots 0.01\%$	0% / 0.001528%	0% / 0.001403%	0.002005% / 15.28%
$0.01 \ldots 0.001\%$	0.000016% / 0.1647%	0% / 0.1444%	5.717% / 39.87%

Table III.7: Comparison of traffic measurement devices with flow IDs defined by the source and destination AS

processing, assuming sufficient SRAM for flow memory and assuming an associative memory. For small flow memory sizes, adding a CAM is quite feasible. Alternatively, one can implement an associative memory using a hash table and storing all flow IDs that collide in a much smaller CAM.

Multistage filters are harder to implement using a network processor because they need multiple stage memory references. However, multistage filters are easy to implement in an ASIC, as the following feasibility study shows. [H01] describes a chip designed to implement a parallel multistage filter with 4 stages of 4K counters each and a flow memory of 3584 entries. The chip runs at OC-192 line speeds. The core logic consists of roughly 450,000 transistors that fit on 2mm x 2mm on a .18 micron process. Including memories and overhead, the total size of the chip would be 5.5mm x 5.5mm and would use a total power of less than 1 watt, which put the chip at the low end of today's IC designs.

III.J Acknowledgements

This chapter is based on [EV03] which is joint work with George Varghese. We thank K. Claffy, D. Moore, F. Baboescu and the anonymous reviewers for valuable comments.

III.K Chapter summary

Motivated by measurements that show that traffic is dominated by a few heavy hitters, this chapter tackles the problem of directly identifying the heavy hitters without keeping track of potentially millions of small flows. Fundamentally, Table III.1 shows that our algorithms have a much better scaling of estimate error (inversely proportional to memory size) than provided by the state of the art Sampled NetFlow solution (inversely proportional to the *square root* of the memory size). On actual measurements, our algorithms with optimizations do several orders of magnitude better than predicted by theory.

However, comparing Sampled NetFlow with our algorithms is more difficult than indicated by Table III.1. This is because Sampled NetFlow does not process every

packet and hence can afford to use large DRAM. Despite this, results in Table III.2 and in Section III.H.2 show that our algorithms are much more accurate for small intervals than NetFlow. In addition, unlike NetFlow, our algorithms provide exact values for long-lived large flows, provide provable lower bounds on traffic that can be reliably used for billing, avoid resource-intensive collection of large NetFlow logs, and identify large flows very fast.

The above comparison only indicates that the algorithms in this chapter may be better than using Sampled NetFlow when the only problem is that of identifying heavy hitters, and when the manager has a precise idea of which flow definitions are interesting. But NetFlow records allow managers to mine *a posteriori* patterns in data they did not anticipate, while our algorithms rely on efficiently identifying stylized patterns that are defined *a priori*. To see why this may be insufficient, imagine that CNN suddenly gets flooded with web traffic. How could a manager realize before the event that the interesting flow definition to watch for is a multipoint-to-point flow, defined by destination address and port numbers?

The last example motivates an interesting question. Is it possible to generalize the algorithms in this chapter to automatically extract flow definitions corresponding to large flows? We answer this question affirmatively in Chapter V. A second open question is to deepen our theoretical analysis to account for the large discrepancies between theory and experiment.

We end by noting that measurement problems (data volume, high speeds) in networking are similar to the measurement problems faced by other areas such as data mining, architecture, and even compilers. For example, Sastry et al. recently proposed using a Sampled NetFlow-like strategy to obtain dynamic instruction profiles in a processor for later optimization [SBS01]. Narayanasamy et al. show [NSSCV03] that multistage filters with conservative update can improve the results of [SBS01]. Thus the techniques in this chapter may be of utility to other areas, and the techniques in these other areas may be of utility to us.

Chapter IV

Bitmap Algorithms for Counting Flows

It is natural to measure the traffic in bytes or packets. In the Internet there is a third useful way to measure traffic: by the number of flows. The number of flows is often used to distinguish malicious traffic such as attacks and scans from normal traffic and to solve other measurement problems. One problem with counting flows is that it needs more than the simple counters that can be used for bytes and packets. In this chapter we present a family of bitmap algorithms that address the problem of counting the number of distinct flows seen on a high speed link. Our algorithms are suitable for hardware implementations with all per packet processing done within a packet arrival time (8 nsec at OC-768 speeds) using only a small number of accesses to limited, fast memory.

A naive flow counting solution that maintains a hash table requires several Mbytes because the number of flows can be above a million. By contrast, our new probabilistic algorithms take very little memory and are fast. The reduction in memory is particularly important for applications that run multiple concurrent counting instances. Besides low memory usage and small per-packet processing, the family of bitmap algorithms gives flexibility to the system designer to customize them to take advantage of special features of applications such as a large number of instances that have very small counts, or prior knowledge of the likely range of the count.

For example, the port scan detection component of the popular intrusion detection system Snort performs the equivalent of identifying and measuring large traffic aggregates we addressed in Chapter III, with the difference that it counts flows, not bytes or packets. We replaced this Snort component with one of our new algorithms. This reduced memory usage on a ten minute trace from 50 Mbytes to 5.6 Mbytes while maintaining a 99.77% probability of alarming on a scan within 6 seconds of when the large-memory algorithm would for very slow scans and much more promptly for aggressive scans. The best known prior algorithm (probabilistic counting[FM85]) takes 4 times more memory on port scan detection and 8 times more on a measurement application.

IV.A Introduction

Internet links operate at high speeds, and past trends predict that these speeds will continue to increase rapidly. Routers that operate at up to OC-768 speeds (40 Gigabits/second) are currently being developed and there is a need for intrusion detection devices that work at high speeds. While the main bottlenecks (e.g., lookups, classification, quality of service) in a traditional router are well understood, what are the corresponding functions that should be hardwired in the brave new world of security and measurement? Ideally, we wish to abstract out functions that are common to several security and measurement applications. We also wish to study efficient algorithms for these functions, especially those with a compact hardware implementation.

Toward this goal, this chapter isolates and provides solutions for an important problem that occurs in various networking applications: *counting the number of active flows among packets received on a link during a specified period of time.* A *flow* is defined by a set of header fields; two packets belong to distinct flows if they have different values for the specified header fields that define the flow. For example, if we define a flow by source and destination IP addresses, we can count the number of distinct source-destination IP address pairs seen on a link over a given time period. Our algorithms measure the number of active flows using a very small amount of memory that can easily be stored in on-chip SRAM or even processor registers. By contrast, naive algorithms described below would require massive amounts of memory necessitating the use of slow

DRAM.

For example, a naive method to count source-destination pairs would be to keep a counter together with a hash table that stores all the distinct 64 bit source destination address pairs seen thus far. When a packet arrives with source and destination addresses say $< S, D >$, we search the hash table for $< S, D >$; if there is no match, the counter is incremented and $< S, D >$ is added to the hash table. Unfortunately, given that backbone links can have up to a million flows [FP99] today, this naive scheme would minimally require 64 Mbits of high speed memory.[1] Such large SRAM memory is expensive or not feasible for a modern router.

There are more efficient general-purpose algorithms for counting the number of distinct values in a multiset. In this chapter we not only present a general-purpose counting algorithm – *multiresolution bitmap* – that has better accuracy than the best known prior algorithm, probabilistic counting [FM85], but introduce a whole family of counting algorithms that further improve performance by taking advantage of particularities of the specific counting application. Our *adaptive bitmap*, using the fact that the number of active flows doesn't change very rapidly, can count the number of distinct flows on a link that contains anywhere from 0 to 100 million flows with an average error of less than 1% using only 2 Kbytes of memory. Our *triggered bitmap*, optimized for running multiple concurrent instances of the counting problem, many of which have small counts, is suitable for detecting port scans and uses even less memory than running adaptive bitmap on each instance.

IV.A.1 Problem Statement

A flow is defined by an *identifier* given by the values of certain header fields.[2] The problem we wish to solve is counting the number of distinct flow identifiers (flow IDs) seen in a specified *measurement interval*. For example, an intrusion detection system looking for port scans could count for each active source address the flows defined by destination IP and port and suspect any source IP that opens more than 3 flows in 12 seconds of performing a port scan. Other applications such as packet scheduling could

[1] It must at least store the flow identifier, which in this example is 64 bits, for each of a million flows.

[2] We can also generalize by allowing the identifier to be a *function* of the header fields (e.g., using prefixes instead of addresses, based on routing tables).

prefer an alternate way of defining the number of active flows without using measurement interval: consider active the flows that have at least one packet in a queue that packets are added to and removed from dynamically. In this chapter we mainly focus on the definition based on measurement intervals.

Also, while many applications define flows at the granularity of TCP connections, one may want to use other definitions. For example when detecting DoS attacks we may wish to count the number of distinct sources, not the number of TCP connections. Thus in this chapter we use the term flow in this more generic way.

As we have seen, a naive solution using a hash table of flow IDs is accurate but takes too much memory. In high speed routers it is not only the cost of large, fast memories that is a problem but also their power consumption and the board space they take up on line cards. Thus, we seek solutions that use a very small amount of memory and have high accuracy. Usually there is a tradeoff between memory usage and accuracy, but we want to find algorithms where these tradeoffs are favorable. Also, since at high speeds per-packet processing time is very limited, it is important that the algorithms use few memory accesses per packet. We describe algorithms that use only 1 or 2 memory accesses[3] and are simple enough to be implemented in hardware.

IV.A.2 Motivation

Why is information about the number of flows useful? We describe five possible categories of use.

Detecting port scans: Intrusion detection systems warn of port scans when a source opens too many connections within a given time.[4] The widely deployed Snort intrusion detection system (IDS) [R99] uses the naive approach of storing a record for each active connection. This is an obvious waste since most of the connections are not part of a port scan. Even for actual port scans, if the IDS only reports the number of connections we don't need to keep a record for each connection. Since the number of sources can be very high, it is desirable to find algorithms that count the number of

[3]Actually, larger numbers of memory accesses are perfectly feasible at high speeds using SRAM and pipelining, but this increases the cost of the solution.

[4]While distributed port scans are possible, they are harder because the attacker has to control many endhosts it can scan from. If the number of hosts is not very large, each will have to probe many port-destination combinations, thus running the risk of being detected.

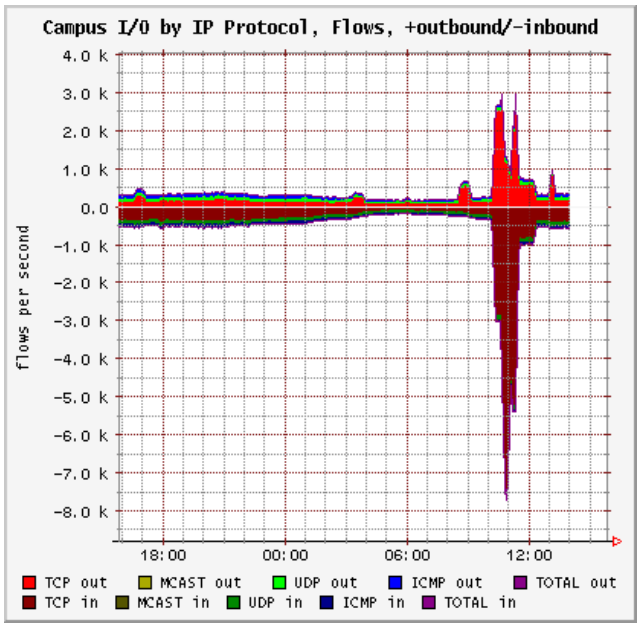


Figure IV.1: The flow count provided by Dave Plonka's FlowScan is used to detect denial of service attacks.

connections of each source using little memory. Further, if an algorithm can distinguish quickly between suspected port scanners and normal traffic, the IDS need not perform expensive operations (e.g., logging) on most of the traffic, thus becoming more scalable in terms of memory usage and speed. This is particularly important in the context of the recent race to provide wire-speed intrusion detection [CW00].

Detecting denial of service (DoS) attacks: FlowScan, by David Plonka [P00], is a popular tool for visualizing network traffic. It uses the number of active flows (see Figure IV.1) to detect ongoing denial of service attacks. While this works well at the edge of the network (i.e., the link between a large university campus and the rest of the Internet), it doesn't scale to the core. Also it relies on massive intermediate data (NetFlow) to compute compact results – could we obtain the useful information more directly? Mahajan et al. propose a mechanism that allows backbone routers to limit the effect of (distributed) DoS attacks [MBFI+01]. While the mechanism assumes that these routers can detect an ongoing attack, they do not give a concrete algorithm for it. Chapter III presents algorithms that can detect destination addresses or prefixes that

receive large amounts of traffic. While these can identify the victims of attacks it also gives many false positives because many destinations have large amounts of legitimate traffic. To differentiate between legitimate traffic and an attack we can use the fact that some DoS tools use fake source addresses chosen at random.[5] If for each suspected victim we count the number of sources of packets that come from some networks known to be sparsely populated, a large count is a strong indication that a DoS attack is in progress.

General measurement: Counting the number of active connections and the number of connections associated with each source and destination IP address is part of the CoralReef [KMKL+01] traffic analysis suite. Other ways of counting the number of distinct values in given header fields can also provide useful data. One could measure the number of sources using a protocol version or variant to get an accurate image of protocol deployment. Alternatively, by counting the number of connections associated with each of the protocols generating significant traffic we can compute the average connection length for each protocol, thus getting a better view of its behavior. Dimensioning the various caches in routers (packet classification caches, multicast route caches for source-group (S-G) state, and ARP caches) also benefits from prior measurements of typical workload.

Estimating the spreading rate of a worm: From Aug 1 to Aug 12 2001, while trying to track the Code Red worm [M01], collecting packet headers for Code Red traffic on a /8 network produced 0.5 GB per hour of compressed data. In order to determine the rate at which the virus was spreading, it was necessary to count the number of distinct Code Red sources[6] passing through the link. This was actually done using a large log and a hash table, which was expensive in time and also inaccurate (because of losses in the log).

Packet scheduling: Many scheduling algorithms try to ensure that all flows can send at the current "fair share" of the available bandwidth. At high speeds it is not feasible to keep per-flow state. While there are scheduling algorithms that compute the fair share without using per-flow state (e.g., CSFQ [SSZ98], XCP [KHR02]), they

[5] If the attack uses a small number of source addresses then it can be easily filtered out once those addresses are identified. Identifying those addresses can be done using techniques from Chapter III because those few source addresses must send a lot of traffic each for the attack to be effective.

[6] Counting the number of active sources is algorithmically equivalent to counting the number of active flows.

require explicit cooperation of edge routers or end hosts. Being able to count the number of distinct flows that have packets in the queue of the router might allow the router to estimate the "fair share" without outside help and could lead to scheduling algorithms that are less vulnerable to misbehaving end hosts or edge routers.

Thus, while counting the number of flows is usually insufficient by itself, it can provide a useful building block for complex tasks that range from detecting DoS attacks to fair packet scheduling.

IV.B Related work

The networking problem of counting the number of distinct flows has a well-studied equivalent in the database community: counting the number of distinct database records (or distinct values of an attribute). Thus, the major piece of related work is a seminal algorithm called *probabilistic counting*, due to Flajolet and Martin [FM85], introduced in the context of databases. We use probabilistic counting as a base to compare our algorithms against. Whang et al. address the same problem and propose an algorithm [WVT90] that is equivalent to the simplest algorithm we describe (direct bitmap).

The insight behind probabilistic counting is to compute a metric of how uncommon a certain record is and keep track of the most uncommon record seen. If the algorithm sees very uncommon records, it concludes that the number of records is large. More precisely, for each record the algorithm computes a hash function that maps it to an L bit string (L is configurable). It then counts the number of consecutive zeroes starting from the least significant position of the hash result. If the algorithm sees records that hash to values ending in 0, 1 and 2 zeroes it concludes that the number of distinct records was $c2^2$ (c is a statistical correction factor close to 1), if it also sees hash values ending in 3 zeroes it estimates $c2^3$ and so on. This basic form can have an accuracy of at most 50% because possible estimates are a factor of 2 from each other. By dividing the hash values into *nmap* groups (*nmap* is configurable), and running a separate instance of the basic algorithm for each group and averaging over the estimates for the count provided by each of them, probabilistic counting reduces the error of its final estimate.

We describe a family of algorithms that each outperforms probabilistic counting by an order of magnitude by exploiting application-specific characteristics.

In networking, there are general-purpose traffic measurement systems such as Cisco's NetFlow [CN] or LFAP [RL] that report per-flow records for very fine-grained flows. This is useful for traffic measurement. The information can be used to count flows (and this is what FlowScan [P00] does), but is not optimized for such a purpose. Besides the large amount of memory needed, in modern, high-speed routers updating state on every packet arrival is infeasible at high speeds. Ideally, such state should be in high speed SRAM (which is expensive and limited) to allow wire-speed forwarding.

To be able to keep state in cheap, slow DRAM, for high speeds NetFlow uses: only the sampled packets result in updates to the flow cache that keeps the per flow state. This affects the accuracy of the measurement data. Sampling works reasonably for estimating the traffic sent by large flows or large traffic aggregates, but has extremely poor accuracy for estimating the number of flows. This is because uniform sampling produces more samples of flows that send more traffic, thereby biasing any simple estimator that counts the number of flows in the sample and applies a correction.

Duffield et al. present two scalable methods for counting the number of active TCP flows based on samples of the traffic [DLT02]. They rely on the fact that TCP turns the SYN flag on only for the packets starting a connection. The estimates are based on counts of the number of flows with SYN packets and the number of flows with non-SYN packets in the sampled data. While this is a good solution for TCP connections it cannot be applied to UDP or when we use a different definition for flows (e.g., when looking at protocol deployment statistics, we define a flow as all packets with the same source IP). Also, counting flows in the sampled data can still be a memory-consuming operation that needs to be efficiently implemented.

The Snort [R99] intrusion detection system (IDS) uses a memory-intensive approach similar to NetFlow to detect port scans: it maintains a record for each active connection and a connection counter for each source IP address. More elaborate algorithms have been used in other settings. When controlling the medium access in wireless networks, some protocols rely on an estimate of the number of senders. The GRAP protocol [YC96] uses techniques equivalent to our direct bitmap and virtual bitmap

to estimate this number. However GRAP has no equivalent of our more sophisticated multiresolution, adaptive, or triggered bitmap algorithms.

IV.C A family of counting algorithms

Our family of algorithms for estimating the number of active flows relies on updating a bitmap at run time. Different members of the family have different rules for updating the bitmap. At the end of the measurement interval (1 second, 1 minute, or even 1 hour), the bitmap is processed to yield an estimate for the number of active flows. Since we do not keep per-flow state, all of our results are estimates. However, we prove analytically and show through experiments on traces that our estimates are close to the actual values. The family contains three core algorithms and four derived algorithms. Even though the first two core algorithms (direct and virtual bitmap) were invented previously, we present them here because they form the basis of our new algorithms (multiresolution, adaptive, and triggered bitmaps and flow sample and hold), and because we present new applications in a networking context (as opposed to a database or wireless context).

We start in Section IV.C.1 with the first core algorithm, *direct bitmap*, that uses a large amount of memory. Next, in Section IV.C.2 we present the second core algorithm called *virtual bitmap* that uses sampling over the flow ID space to reduce the memory requirements. While virtual bitmap is extremely accurate, it needs to be tuned for a given anticipated range of the number of flows. We remove the "tuning" restriction of virtual bitmap with our third algorithm called *multiresolution bitmap*, described in Section IV.C.3, at the cost of increased memory usage. Next we describe four derived algorithms for counting the number of active flows. *Triggered bitmap* described in Section IV.C.4 combines direct bitmap and multiresolution bitmap to reduce the total amount of memory used by multiple instances of flow counting when most of the instances count few flows. Applying the ideas behind virtual bitmap to the sample and hold algorithm gives us *flow sample and hold*, an algorithm suited for identifying sources or destinations with a large number of flows. In Section IV.C.6 we show how we can adapt the core algorithms to the alternate definition of active flows: the ones that have

packets in a queue that supports arbitrary additions and removals (not those that send any packets during a fixed measurement interval). Since our fourth algorithm *adaptive bitmap*, exemplifies the paradigm of adapting the traffic measurement device to the traffic mix, we do not describe it here, but in Section VI.A.3. In this section we only describe the algorithms; we leave an analysis of the algorithms to Section IV.D.

IV.C.1 Direct bitmap

The direct bitmap is a simple algorithm for estimating the number of flows. We use a hash function on the flow ID to map each flow to a bit of the bitmap. At the beginning of the measurement interval all bits are set to zero. Whenever a packet comes in, the bit its flow ID hashes to is set to 1. Note that all packets belonging to the same flow map to the same bit, so each flow turns on at most one bit irrespective of the number of packets it sends.

We could use the number of bits set as our estimate of the number of flows, but this is inaccurate because two or more flows can hash to the same bit. In Section IV.D.1, we derive a more accurate estimate that takes into account hash collisions.[7] Even with this better estimate, the algorithm becomes very inaccurate when the number of flows is much larger than the number of bits in the bitmap and the bitmap is almost full. The reason it becomes inaccurate when the bitmap is almost full is that the number of bits that are not set decreases exponentially with the number of flows. The difference between the number of flows expected to leave 1 bit unset (i.e. the expected number of bits not set is one) and the number of flows expected to leave 2 bits unset is much larger than between the number of flows expected to leave 10 and 11 bits unset. However because of the randomness of the hashing, the number flows we expect to leave 1 bit unset is quite likely to leave 2 or leave none. The only way to preserve accuracy is to have a bitmap size that scales almost linearly with the number of flows, which is often impractical.

[7] We assume in our analysis that the hash function distributes the flows randomly. In an adversarial setting, the attacker who knows the hash function could produce flow identifiers that produce excessive collisions thus evading detection. This is not possible if we use a random seed to our hash function.

IV.C.2 Virtual bitmap

The virtual bitmap algorithm reduces the memory usage by storing only a small portion of the big direct bitmap one would need for accurate results (see Figure IV.2) and extrapolating the number of bits set. This can also be thought of as sampling the flow ID space. The larger the number of flows the smaller the portion of the flow ID space we cover. Virtual bitmap generalizes direct bitmap: direct bitmap is a virtual bitmap which covers the entire flow ID space.

Unfortunately, a virtual bitmap does require tuning the "sampling factor" based on prior knowledge of the number of flows. If it differs significantly from what we configured the virtual bitmap for, the estimates are inaccurate. If the number of flows is too large, the virtual bitmap fills up and has the same accuracy problems as an underdimensioned direct bitmap. If the number of flows is too small we have another problem: say the virtual bitmap covers 1% of the flow ID space and there are 50 active flows - if none of them hashes to the virtual bitmap, the algorithm will suppose the number of flows is 0, if 1 hashes, the algorithm will estimate 100, but it will never estimate 50. The optimal sampling factor obtains the best tradeoff between "collision errors" and "extrapolation errors".

While, in general, one wants an algorithm that is accurate over a wider range, we note that even an unadorned virtual bitmap is useful. For example, consider a security application where we wish to trigger an alarm when the number of flows crosses a threshold. The virtual bitmap can be tuned for this threshold and uses less memory than other algorithms that are accurate not just around the threshold, but over a wider range for the number of flows.

In Section IV.D we derive formulae for the average error of the virtual bitmap estimates. The analysis also provides insight for choosing the right sampling factor. Perhaps surprisingly, the analysis also indicates that the average error depends only on the number of bits and not on the number of flows as long as the sampling factor is set to an optimal value. For example with 215 bytes the average error is 3%.

IV.C.3 Multiresolution bitmap

The virtual bitmap is simple to implement, uses little memory, and gives very accurate results, but requires us to know in advance a reasonably narrow range for the number of flows. An immediate solution to this shortcoming is to use many virtual bitmaps, each using the same number of bits of memory, but different sampling factors, so that each is accurate for a different range of the number of active flows (different "resolutions"). The union of all these ranges is chosen to cover all possible values for the number of flows. When we compute our estimate, we use the virtual bitmap that is most accurate based on a simple rule that looks at the number of bits set. The "lowest resolution" bitmap is a direct bitmap that works well when there are very few flows. The "higher resolution" bitmaps cover a smaller and smaller portion of the flow ID space and work well when the number of flows is larger. The problem with the naive approach of using several virtual bitmaps of differing granularities is that instead of updating one bitmap for each packet, we need to update several, causing more memory accesses.

The main innovation in multiresolution bitmap is to maintain the advantages of multiple bitmaps configured for various ranges while performing a *single update* for each incoming packet. Figure IV.2 illustrates the direct bitmap, virtual bitmap, multiple bitmaps and multiresolution bitmap. Before explaining how the multiresolution bitmap works, it can help to switch to another way of thinking about how the virtual bitmap operates. We can consider that instead of generating an integer, the hash function covers a continuous interval. The virtual bitmap covers a portion of this interval (the ratio of the sizes of the interval covered by the virtual bitmap and the entire interval is the sampling factor of the virtual bitmap). We divide the interval corresponding to the virtual bitmap into equal-sized sub-intervals, each corresponding to a bit. A bit in the virtual bitmap is set to 1 if the hash of the incoming packet maps to the sub-interval corresponding to the bit. The multiple bitmaps solution is shown below the virtual bitmap solution in Figure IV.2.

A multiresolution bitmap is essentially a combination of multiple bitmaps of different "resolutions", such that a single hash is computed for each packet and only the highest resolution bitmap it maps to is updated. Thus each bitmap loses a portion of its bits which are covered by higher resolution bitmaps. But those bits can easily be

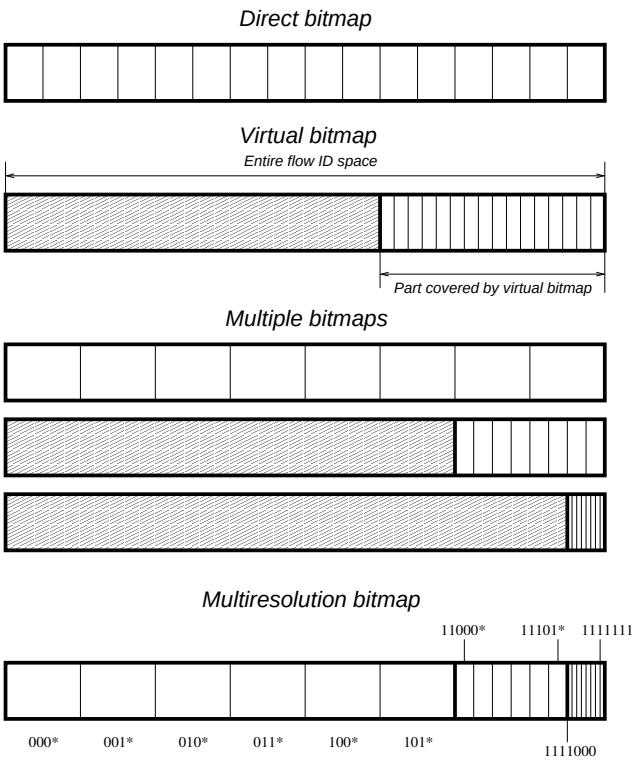


Figure IV.2: The multiresolution bitmap from this example uses a single 7-bit hash function to decide which bit to map a flow to. It gives results no less accurate than the 3 virtual bitmaps, thus covering a wide range for the number of flows, but it performs a single memory update per packet. Note that all the unfilled "tiles" from these bitmaps, despite their different sizes, represent one bit of memory.

recovered later (during the analysis phase) from the finer grained bitmaps by OR-ing together the bits in the higher resolution bitmaps that correspond to individual bits in the lower resolution bitmap. We call these regions with different resolutions components of the multiresolution bitmap. When we compute the estimate, based on the number of bits set in each component, we choose one of them as "base", estimate the number of flows hashing to it and all finer components, and extrapolate.

In Section IV.D.3 we answer questions such as: how many bits should each component have, how many components do we need and what is the best ratio between the resolutions of neighboring components? In Appendix E.4 we show that multiresolution bitmaps are easy to implement even in hardware that can keep up with line speeds.

Also, we compare our multiresolution bitmap to probabilistic counting, showing that while both algorithms use nearly identical hashes to set bits, they interpret the data *very* differently, thus the differences in the accuracy of the results.

IV.C.4 Triggered bitmap

Consider the concrete example of detecting port scans. If one used a multiresolution bitmap per active source to count the number of connections, the multiresolution bitmap would need to be able to handle a large number of connections because port scans can use very many connections. The size of such a multiresolution bitmap can be quite large. However, most of the traffic is not port scans and most sources open only one or two connections. Thus using a large bitmap for each source is wasteful.

The triggered bitmap combines a very small direct bitmap with a large multiresolution bitmap. All sources are allocated a small direct bitmap. Once the number of bits set in the small direct bitmap exceeds a certain trigger value, a large multiresolution bitmap is allocated for that source and it is used for counting the connections from there on. Our estimate for the number of connections is the sum of the flows counted by the small direct bitmap and the multiresolution bitmap. This way we have accurate results for all sources but only pay the cost of a large multiresolution bitmap for the sources that open many connections.

As described so far, this algorithm introduces a subtle error that makes a small change necessary. If a flow is active both before and after the large multiresolution bitmap is allocated, it gets counted by both the direct bitmap and the multiresolution bitmap. Only using the multiresolution bitmap for our final estimate is not a solution either because then we would not count the flows that were active only before the multiresolution bitmap was allocated. To avoid this problem we change the algorithm the following way: after the multiresolution bitmap is allocated, we only map to it those flows that do not map to one of the bits already set in the direct bitmap. This way if the flows that set the bits in the direct bitmap send more packets, they will not influence the multiresolution bitmap. It's true that the multiresolution bitmap doesn't catch all the new flows, just the ones that map to one of the bits not set in the direct bitmap. This is equivalent to the "sampling factor" of the virtual bitmap and we can compensate for

it (see Section IV.D.1).

IV.C.5 Flow sample and hold

While triggered bitmap reduces memory usage significantly, it still allocates memory for each active source. As shown in Section III.E.1, sample and hold creates entries with high probability for the sources that send many packets (we use packet sample and hold or PSH to distinguish sample and hold with sampling probabilities equal for all packets from the version presented in Section III.D.1, where sampling probability depends on packet size) and has a low probability of allocating an entry for sources with few packets. Can we use sample and hold to create entries only for the sources with many flows? A simple line of thought seems to confirm that PSH could also be used to identify source IP addresses with many flows, since a source can have many flows only if it has at least as many packets in the traffic, but as the following example shows, PSH cannot always achieve this goal. Let's assume that we have a traffic mix consisting of peer to peer traffic moving 1.5 megabyte songs (consisting of 1000 maximum sized packets) from random IP addresses to random IP addresses along with a few port scanners scanning at a rate of 50 destinations per 5 minute interval. Of course we want to detect the port scanners, because those sources have the largest number of active flows. In 5 minutes there is room for approximately 60,000 1.5 megabyte files on an OC-48. Say we only have space for 50,000 entries in the source IP table. Thus if we use PSH, we can sample at most one packet in 1200 to ensure that the table doesn't fill up. At this packet sampling rate the probability of catching a port scanner is approximately 1/24. We are not catching the port scanners because the probability for a source IP getting an entry in the table depends on the number of packets it sends, not on the number of flows it has. In general, traffic mixes dominated by very many sources with few flows that have many packets will make it hard for PSH to catch early enough the sources that have many flows with few packets in each. While admittedly the traffic mix from our example above is not typical, but if we want to build a system that is robust in the face of changes in the traffic mix, we need an algorithm that doesn't have such an unpalatable failure mode.

Flow sample and hold (FSH) is similar to packet sample and hold (PSH), but it

uses a sampling function that makes it more likely for sources with many flows to get an entry in the source table: it samples the flow space the same way virtual bitmap does, instead of sampling packets independently, irrespective of their flow ID. We hash the flow identifiers of packets; for the packets whose identifiers fall into a certain subinterval of that hash space, we create an entry in the source IP table. Say the hash on the flow ID produces 32-bit hash values. We keep a variable f that controls the rate at which flows are sampled. If the hash value is below f, an entry is created. Setting f to $2^{32} - 1$ amounts to FSH creating an entry for each flow. As we use lower values for f, fewer entries are created. The flow sampling probability of FSH is $f/2^{32}$. Notice that all packets belonging to a flow will have the same hash value, thus each flow has the same probability of triggering the creation of an entry for its source IP, irrespective how many packets it has. Furthermore, as the number of flows a source has increases the probability of it not getting an entry that tracks it decreases exponentially. Therefore sources with many flows will get entries early on and the flow counter in that entry will count all further flows initiated after the entry is created (and even the old ones if they send more packets). For the example above, let's say we use flow sampling with a sampling probability of 0.1. The probability that a flow scanner gets an entry is $(1 - 0.9^{50}) \approx 99.5\%$. At the same time the sources of peer to peer traffic that have one flow each will have a probability of 0.1 of receiving an entry. Thus the peer to peer traffic will add only around 60,000*0.1= 6,000 entries to the source IP table.

Can flow sample and hold replace packet sample and hold? Is it guaranteed to catch sources sending many packets (or bytes)? The answer is no. Imagine that we also have in the traffic mix from above someone only sending a 100 megabyte file through a single TCP connection. This is by far the largest sender, but if the single TCP connection doesn't hash onto the portion of the hash space that triggers addition to the source IP table (and this will happen with a probability of 90% using the parameters above), we will not create an entry and thus ignore this important source. The obvious answer for a system that aims to detect the sources that send many packets and also those that have many flows is to use both PSH and FSH to populate the tables with entries.

IV.C.6 Handling packet removals

We mentioned earlier that counting the number of flows that have packets in the queue of a router can help determine the "fair share" used by the scheduling algorithm. In this case, we need to not only handle the case of new packets arriving but also the case of packets getting removed. Direct bitmap, virtual bitmap and multiresolution bitmap can be easily modified to handle this case by replacing every bit with a counter. The width of the counters is given by the maximum number of packets the queue can accommodate (which also puts a limit on the number of distinct flows that can have packets in the queue). When the queue is empty, all counters are 0. When a new packet arrives, the counter it maps to is incremented. When a packet is removed from the queue, the counter is decremented. We use the number of counters with value zero to compute our estimate of the number of active flows exactly the same way we use the number of zero bits to estimate the number of active flows in a measurement interval. A counter will be zero if and only if no active flows map to it.

IV.C.7 Combining or comparing multiple measurement intervals

If we estimate the number of active flows for 5 second measurement intervals, we might also be interested in the number of active flows over a longer, say one minute, interval. If all flows are short-lived than the answer will be the sum of the counts for the twelve 5 second intervals, but if there is a significant number of long-lived flows, the answer will be much lower. We could solve the problem by running in parallel an instance of the algorithm that uses a one minute measurement interval but that would double the per packet processing. Another solution is to keep the bitmaps of the small intervals (generated using the same hash function) and perform a logical "or" over them. The bits set are going to be exactly those that an instance with a one-minute measurement interval would have turned on, so we can use this bitmap to estimate the number of flows active over the one minute interval.

This simple technique can be applied in other ways too. For example, if we have the bitmaps for two intervals we can compute the number of new flows that appeared in the second one by computing the number of flows active over the union of both intervals and subtracting the number of flows active in the first one. Another application is

staggered intervals: often we don't want to divide up the time into measurement intervals, but to always have an estimate for the number of flows over the last say minute. By using say 5 second intervals and keeping the last 11 intervals as well we can always compute the number of flows active since the start of the first interval.

IV.D Algorithm Analysis

In this section we provide the analyses of the statistical behavior of the bitmaps used by our algorithms. We focus on three types of results. In Section IV.D.1, we derive formulae for estimating the number of active flows based on the observed bitmaps. In Section IV.D.2, we analytically characterize the accuracy of the algorithms by deriving formulae for the average error of the estimates. In Section IV.D.3, we use the analysis to derive rules for dimensioning the various bitmaps so that we achieve the desired accuracy over the desired range for the number of flows.

IV.D.1 Estimate Formulae

Direct bitmap: To derive a formula for estimating the number of active flows for a direct bitmap, we have to take into account collisions. Let b be the size of the bitmap. The probability that a given flow hashes to a given bit is $p = 1/b$. Assuming that n is the number of active flows, the probability that no flow hashes to a given bit is $p_z = (1-p)^n \approx (1/e)^{n/b}$. By linearity of expectation, this formula gives us the expected number of bits not set at the end of the measurement interval $E[z] = bp_z \approx b(1/e)^{n/b}$. If the number of zero bits is z, Equation IV.1 gives our estimate $\widehat{n}$ for the number of active flows. Whang et al. also show that this is the maximum likelihood estimator for the number of active flows [WVT90].

$$\widehat{n} = b \ \ln\left(\frac{b}{z}\right) \tag{IV.1}$$

Virtual bitmap: Let α be the "sampling factor" (the ratio of the sizes of the interval covered by the virtual bitmap b and the entire hash space h). The probability for a given flow to hash to the virtual bitmap is equal to the sampling factor $p_v = \alpha = b/h$. Let m be the number of flows that actually hash to the virtual bitmap. Its probability

distribution is binomial with an expected value of $E[m] = \alpha n$. We can use Equation IV.1 to estimate m and based on that we obtain Equation IV.2 for the estimate of the number of active flows n.

$$\widehat{n} = \frac{1}{\alpha}\, b \ln\left(\frac{b}{z}\right) = h \ln\left(\frac{b}{z}\right) \tag{IV.2}$$

Multiresolution bitmap: The multiresolution bitmap is a combination of many components, each tuned to provide accurate estimates over a particular range. When we compute our estimate we don't know in advance which component is the one that provides the most accurate estimate (we call this the base component). As we will see in Section IV.D.2, we obtain the smallest error by choosing as the base component the coarsest component that has no more than set_{max} bits (lines 1 to 5 in Figure IV.3). set_{max} is a precomputed threshold based on the analysis from Section IV.D.2. Once we have the base component, we estimate the number of flows hashing to the base and all the higher resolution ones using Equation IV.1 and add them together (lines 13 to 17 in Figure IV.3). To obtain the result we only need to perform the multiplication corresponding to the sampling factor (lines 18 and 19). Other parameters used by this algorithm are the ratio k between the resolutions of neighboring components and b_{last}, the number of bits in the last component (which is different from b).

Triggered bitmap: If the triggered bitmap did not allocate a multiresolution bitmap, we simply use the formula for direct bitmaps (Equation IV.1). Let's use g for the number of bits that have to be set in the direct bitmap before the multiresolution bitmap is allocated, and d for the total number of bits in the direct bitmap. If the multiresolution bitmap is deployed, we use the algorithm from Figure IV.3 to compute the number of flows hashing to the multiresolution bitmap, multiply that by $d/(d-g)$ and add the estimate of the direct bitmap.

IV.D.2 Accuracy

To determine the accuracy of these algorithms we look at the standard error of our estimate $\widehat{n}$, that is the standard deviation of the ratio $\widehat{n}/n$. We also refer to this quantity as the average (relative) error $SD[\widehat{n}/n] = SD[\widehat{n}]/n$. One parameter that is useful in these analyses is the flow density ρ defined as the average number of flows that

```
ESTIMATEFLOWCOUNT
1   base = c − 1
2   while base > 0 and bitsSet(component[base]) ≤ set_max
3       base = base − 1
4   endwhile
5   base = base + 1
6   if base == c and bitsSet(component[c]) > setlast_max)
7       if bitsSet(component[c]) == b_last
8           return "Cannot give estimate"
9       else
10          warning "Estimate might be inaccurate"
11      endif
12  endif
13  m = 0
14  for i = base to c − 1
15      m = m + b ln(b/bitsZero(component[i]))
16  endfor
17  m = m + b_last ln(b_last/bitsZero(component[c]))
18  factor = k^(base−1)
19  return factor ∗ m
```

Figure IV.3: Algorithm for computing the estimate of the number of active flows for a multiresolution bitmap. We first pick the base component that gives the best accuracy then add together the estimates for the number of flows hashing to it and all higher resolution components and finally extrapolate.

hash to a bit.

Direct bitmap: While our formula for estimating the number of active flows accounts for the expected collisions, it doesn't always give exact results because the number of collisions is random. Equation IV.3 approximates the average error of a di-

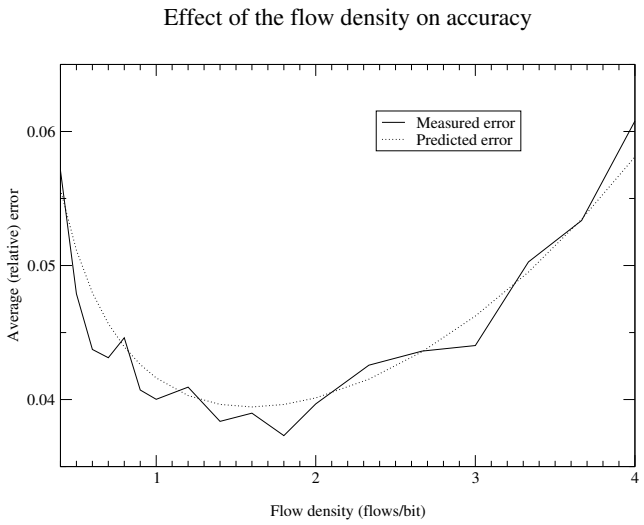


Figure IV.4: When the flow density is too low, the "sampling error" takes over; when it is too high, "collision error" is the main factor. We get the best accuracy for a flow density of around $\rho = 1.6$. The estimate from Equation IV.4 matches well the experimental results, being slightly conservative (larger). See Section IV.E.1 for details on the experiment that produced this result.

rect bitmap based on the Taylor expansion of Equation IV.1 as derived by Whang et al. [WVT90]. The result is not exact because less significant terms of the Taylor expansion were omitted. Whang et al. also show that the approximation does not lead to serious inaccuracies for configurations one expects to see in practice. They also show that the distribution of the number of bits set is asymptotically normal so errors much larger than the standard error are very unlikely [WVT90]. For example, for a direct bitmap configured to operate at an average error of 10% for flow densities up to 2, the value of the average error we get by including the next term of the Taylor series is only 2% away from the approximation (i.e., the actual average error can be at most 10.2% instead of 10%). The inaccuracy introduced by the approximation decreases further as the number of bits increases.

$$SD\left[\frac{\widehat{n}}{n}\right] \approx \frac{\sqrt{e^{\rho} - \rho - 1}}{\rho\sqrt{b}} \tag{IV.3}$$

Virtual bitmap: Besides the randomness in the collisions, there is another

source of error for the virtual bitmap: we assume that the ratio between the number of flows that hash to the physical bitmap and all flows is exactly the sampling factor, while due to the randomness of the process the number can differ. In Appendix E.1 we analyze these two errors and how they interact. Equation IV.4 takes into account their cumulative effect on the result. When the flow density is too large, the error increases exponentially because of the collision errors. When it is too small, the error increases as the sampling errors take over. Our analysis also shows that the terms ignored by the approximations do not contribute significantly and that the bound is tight. Figure IV.4 presents a typical result comparing the measured average error from simulations on traces of actual traffic to the value from Equation IV.4.

$$SD\left[\frac{\widehat{n}}{n}\right] \lessapprox \frac{\sqrt{e^{\rho}-1}}{\rho\sqrt{b}} \tag{IV.4}$$

Multiresolution bitmap: To compute the average error of the estimate of the multiresolution bitmap, we should take into account separately the collision errors of all components finer than the base. This would result for a different formula for each component that would be used as base. Equation IV.5 is a slightly weaker bound that holds for all components but the last one as long as the number of bits in the last component b_{last} is large enough. The details of its derivation can be found in Appendix E.1. Equation IV.5 bounds quite tightly the average error for a normal component. For the last component of the multiresolution bitmap we use Equation IV.4 directly.

$$SD\left[\frac{\widehat{n}}{n}\right] \lessapprox \frac{\sqrt{\frac{k-1}{k}\left(e^{\rho}+e^{\rho/k}-2\right)+e^{\rho/k^2}-1}}{\rho\sqrt{\frac{bk}{k-1}}} \tag{IV.5}$$

IV.D.3 Configuring the bitmaps

In this section we address the configuration details and implicitly the memory needs of the bitmap algorithms. All measurement results are in Section IV.E. The two main parameters we use to configure the bitmaps are the maximum number of flows one wants them to count, N, and the acceptable average *relative* error, ϵ. We base our computations on the formulas of the previous section.

Algorithm	Memory (bits)
Direct bitmap	$< N/\ln(N\epsilon^2+1)$
Virtual bitmap	$1.54413865/\epsilon^2$
Multiresolution bmp.	$0.9186\ln(N\epsilon^2)/\epsilon^2 + ct.$

Table IV.1: The size of the direct bitmap scales sublinearly with N, roughly as $N/\ln(N\epsilon^2+1)$; the size for the virtual bitmap is proportional to the inverse of the square of the average error; and the size of the multiresolution bitmap scales with the logarithm of the number of flows over the square of the average error.

Direct bitmap: If we would keep $\rho = N/b$ constant as N increased, ϵ would improve proportionally to $1/\sqrt{N}$ (which is proportional to $1/\sqrt{b}$). So as N increases the flow density that gives us the desired accuracy also increases. Therefore by ignoring the constant term under the square root in Equation IV.3 we get a tight bound on how b scales. $\epsilon^2 \lessapprox (e^\rho - \rho)/(\rho^2 b)$ so $\epsilon^2 N + 1 \lessapprox e^\rho/\rho < e^\rho$. From here, $\rho > \ln(\epsilon^2 N + 1)$ and thus $b < N/\ln(\epsilon^2 N + 1)$. We claim that for large values of N while this closed form bound is not tight, it is not very far off either. For example for $N = 1,000,000$ and $\epsilon = 10\%$ the bound gives 108,572 bits while the actual value is 85,711 bits. Of course, for configuring a direct bitmap we recommend solving Equation IV.3 numerically for b (with ρ replaced by N/b).

Virtual bitmap: The average error of the virtual bitmap given by Equation IV.4 is minimized by a certain value of the flow density. Solving numerically we get $\rho_{optimal} = 1.593624$ and this corresponds to around 20.3% of the bits of the bitmap being not set. By substituting, we obtain the average error for this "sweet spot" flow density $\epsilon \lessapprox 1.242633756330/\sqrt{b}$. By inverting this we obtain the formula from Table IV.1 for the number of bits of physical memory we need to achieve a certain accuracy. When we need to configure the virtual bitmap as a trigger, we set the sampling factor such that at the threshold the flow density is 1.593624. For this application, if we have 155 bits, the average error of our estimate is at most 10% no matter how large the threshold. If we have 1,716 bits, the average error is at most 3%, and if we have 15,442 bits it is at most 1%. If we want to have at most a certain error for a range of flow counts between N_{min} and N_{max}, we need to solve the problem numerically by finding a $\rho_{min} < \rho_{optimal}$ and a $\rho_{max} > \rho_{optimal}$ so that $\rho_{max}/\rho_{min} = N_{max}/N_{min}$ and ρ_{min} and ρ_{max} produce the same

k	ρ_{min}	ρ_{max}	coefficient $f(k)$	$f(k)/\ln(k)$
2	1.3372	2.6744	0.6367	0.9186
3	0.9750	2.9250	1.0318	0.9392
4	0.7856	3.1426	1.3470	0.9717

Table IV.2: The operating range of the components of the multiresolution bitmap is between ρ_{min} and ρ_{max}. The coefficient and the desired accuracy determine the size of the components $b = f(k)/\epsilon^2$. The larger the ratio between the resolutions of neighboring components k, the wider the range covered by a single component and the larger the component.

error. Once we have these values, we can compute the sampling factor for the virtual bitmap and the number of bits.

Multiresolution bitmap: For the multiresolution bitmap, we have to ensure that the average error doesn't exceed the desired value over the whole range from 0 to N. We divide the range among components. Configuring a component is very much like configuring a virtual bitmap for a range, except we use Equation IV.5. We find two flow densities, ρ_{min} and ρ_{max}, that give the same error under the constraint that $\rho_{max}/\rho_{min} = k$ (k is the ratio between the resolutions of neighboring components). We choose the bitmap size b for the normal components (all except the last one) such that at ρ_{min} and ρ_{max} we get the desired accuracy $b = f(k)/\epsilon^2$, where the coefficient $f(k)$ depends on k. Table IV.2 contains the values of ρ_{min}, ρ_{max} and the coefficient used for determining the bitmap size for three useful values for k. The base component is the one with a flow density between ρ_{min} and ρ_{max}, so the threshold used by the algorithm (Figure IV.3) to select the base component is $set_{max} = b(1 - e^{-\rho_{max}})$.

We can choose the number of components such that the last normal component (the penultimate overall) covers the end of the range N: $c = 2 + \lceil log_k(N/(\rho_{max}b)) \rceil$. The total size of the multiresolution bitmap is $Mem = b * (c-1) + b_{last}$. Thus, ignoring the additive constants, the asymptotic memory usage is $Mem \approx \ln(N\epsilon^2)/\epsilon^2 f(k)/\ln(k)$. By allocating more bits to the last component than what it needs in order to make the penultimate component accurate at ρ_{max}, it can also provide accurate enough estimates, and this allows us to reduce the number of components in the bitmap. The algorithm for computing the optimal configuration is long but not very complicated: it evaluates

Name	No. of flows (min/avg/max)	Length (s)
MAG+	93,437 / 98,424 / 105,814	4515
MAG	99,264 / 100,105 / 101,038	90
COS	17,716 / 18,070 / 18,537	90
IND	1,964 / 2,164 / 2,349	90

Table IV.3: The traces used for our measurements

some choices for b_{last} and c and picks the best one. The full algorithm is presented in Appendix E.2.

The ratio $f(k)/\ln(k)$ gives the asymptotic memory usage for a certain choice of k, and we can see from Table IV.2 that $k = 2$ is the best choice.[8] The algorithm is very easy to implement in hardware if k, $bk/(k-1)$ and b_{last} are powers of two. Under these constraints, sometimes the choice of $k = 4$ gives a smaller memory usage because the size b of the components it needs to achieve the desired average error ϵ "fits better" the powers of two. Therefore when configuring the algorithm for a hardware implementation that has these limitations, it is best to check both values of $k = 2$ and $k = 4$. [9]

IV.E Measurement results

We group our measurements into 4 sections corresponding to the 4 important algorithms presented: virtual bitmap, multiresolution bitmap and triggered bitmap. Part of the measurements are geared toward checking the correctness of the predictions of our theoretical analysis and part are geared toward comparing the performance of our algorithms with probabilistic counting or other existing solutions.

For our experiments, we used the same traces as in Section III.H. We usually set the measurement interval to 5 seconds. In all experiments we defined the flows by the 5-tuple of source and destination IP addresses, ports, and protocol. Table IV.3 gives a summary description of the traces we used. All algorithms used equivalent CRC-based hash functions with random generator functions.

[8] There are some very rare cases when $k = 3$ gives a slightly smaller memory usage. This is because the number of components cannot be fractional and the components for $k = 3$ "fit better" to the given N and ϵ.

[9] We found no set of parameters N,ϵ for which $k = 8$ worked better than both $k = 2$ and $k = 4$

IV.E.1 Virtual bitmap

We performed experiments to check the validity of Equation IV.3 for various configurations on many traces. Figure IV.4 shows a typical result. More results can be found in Appendix F. Our measurements confirm that Equation IV.3 gives a tight and slightly conservative bound on the average error (conservative in the sense that actual errors are usually somewhat smaller than predicted by the formula). The results also confirm that we get the best average error for a virtual bitmap of a given size when the flow density is around $\rho = 1.6$.

We also compare the average error of the virtual bitmap to probabilistic counting using the same amount of memory for a variety of configurations and traces. Because our major contributions are the remaining schemes, we provide here only one sample result. For the COS Trace, using 1,716 bits our analysis predicts an expected error 3%. Over 20 runs, for the 18 measurement intervals, the actual average error (computed as square root of the average of squares) for virtual bitmap is 2.773% with a maximum of 9.467%. This is not just a further confirmation that Equation IV.3 gives a tight bound on the average error, but it also shows that errors much larger than the average error are very unlikely. On the other hand, probabilistic counting configured to handle up to 100,000 flows had an average error of 6.731% with a maximum of 27.336%. While this is an unfair comparison in general (virtual bitmap requires knowing in advance the range of final count values), it does fairly indicate our major message: a problem-specific counting method for a specific problem like threshold detection can significantly outperform a one-size-fits-all technique like probabilistic counting.

IV.E.2 Multiresolution bitmap

This set of experiments compares the average error of the multiresolution bitmap and probabilistic counting. Both algorithms are expected to give accurate estimates over a wide range for the number of active flows. A meaningful comparison is possible if we compare the two algorithms over the whole range for the number of flows. Since our traces have a pretty constant number of flows, we use a synthetic trace for this experiment. We used the actual packet headers from the MAG+ trace to generate a trace that has a different number of flows in each measurement interval: from 10 to

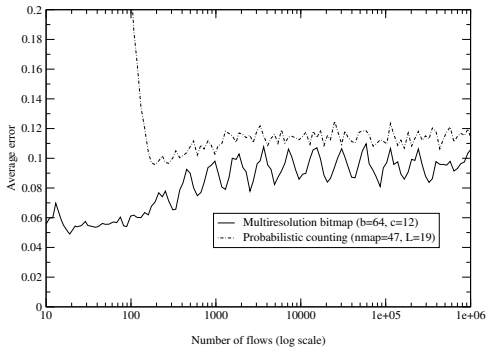


Figure IV.5: Configured for an average error of 10%

1,000,000 in increments of 10% with a jitter of 1% added to avoid any possible effects of "synchronizations" with certain series of numbers.

We ran experiments with multiresolution bitmaps tuned to give an average error of 1%, 3% and 10% for up to 1,000,000 flows and probabilistic counting configured for the same range with the same amount of memory. We had 500 runs for each configuration of both algorithms with different hash functions.

Figures IV.5 to IV.7 show the results of the experiments. We can see that in all three experiments, the average error of the multiresolution bitmap is better than predicted for small values, because we have no "sampling error" when the number of flows is small. We explain the periodic "fluctuations" of average error from figure IV.5 by occasional incorrect choice of the base component. The peaks correspond to where components are least accurate and hand off to each other. The peaks are more pronounced in this figure than the others because due to the small number of bits in each component, it happens more often than not the best component is used as a base for the estimation. In Figure IV.6 and especially in Figure IV.7 there is a visible decrease in the error for the multiresolution bitmap when the number of flows approaches the upper limit. The reason is that the last component is much larger than the normal ones and provides more accurate results.

Probabilistic counting is worse than the multiresolution bitmap, especially for small values. We show in Appendix E.3 that the data collected by the two algorithms is equivalent, so it might be surprising that their accuracies are so different. We attribute

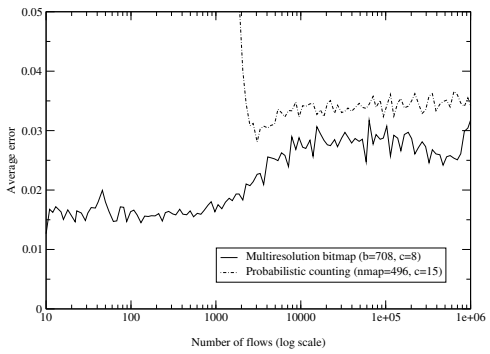


Figure IV.6: Configured for an average error of 3%

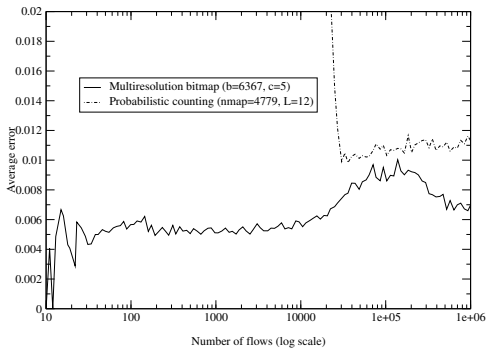


Figure IV.7: Configured for an average error of 1%

the large errors of probabilistic counting for low values to the way it evaluates the collected data. The ability of multiresolution bitmap to be accurate on the low end of the range too can lead to simpler, more robust systems. We attribute the worse error of probabilistic counting for higher values mostly to the suboptimal dimensioning of the algorithm (as recommended in [FM85]).

IV.E.3 Triggered bitmap

So far, all our measurements have focused on one instance of the counting problem to be used as a building block for solving more complex problems. The experiments from this section give a better image of how using our algorithms can affect the resource consumption of an entire system.

We first address port scan detection that uses a large number (one per source) of instances of the counting problem, multiplying the impact of any memory our algorithm can save. We use a definition of a port scan equivalent to the definition in the default Snort configuration: a source is flagged as a port scanner if it has at least 4 connections in a 12 second measurement interval. In the second experiment we extend the measurement interval to 10 minutes to evaluate the algorithms against this more demanding definition. We ignore many of the details of the operation of Snort (e.g., reliance on TCP flags to classify connections) and concentrate on the core task of counting connections.

For the triggered bitmap we chose a configuration that is convenient to implement on a 32-bit machine: a direct bitmap of 4 bytes and a multiresolution bitmap with 11 components of 4 bytes each (except the last one which is 8 bytes). The multiresolution bitmap is allocated after 8 bits are set in the direct bitmap. By our analysis the multiresolution bitmap should ensure an average error of at most 14.1% for up to 43,817 connections and at most 15.5% for up to 175,269 connections.

We compute memory usage of Snort based on the number of sources and connections active during the measurement interval. What we actually use is a not an accurate model of the memory usage of Snort (which uses inefficient structures such as multiple linked lists) but the minimum that any implementation using the naive algorithm would have to allocate: 8 bytes for the IP address and a counter for each source and 9 bytes (destination IP, source port, destination port, type) for the identifier of each active connection. We also compute the memory usage of a solution directly applying probabilistic counting with a configuration similar to our multiresolution bitmap (48 bytes for the algorithm + 4 bytes for the IP address for each source). Our triggered bitmap algorithm consumes 8 bytes for each active source (the IP address + the direct bitmap) plus the additional 48 bytes for the sources that trigger the allocation of the multiresolution bitmap.

Measurement interval	Snort	Prob. count.	Triggered bmp.
12 sec	1,968K	2,474K	381K
600 sec	50,791K	22,876K	5,725K

Table IV.4: The memory usage of port scan detection algorithms (Kbytes)

Setting	Algorithm	Application
General counting	Multiresolution bitmap	Tracking worm populations
Accuracy important only over a narrow range	Virtual bitmap	Triggers (e.g. for detecting DoS attacks)
Count is probably in a narrow range (stationarity)	Adaptive bitmap (Section VI.A.3)	Measurement
Small memory usage as long as count is small	Triggered bitmap	Detecting port scans
Flows dynamically added and deleted	Increment-decrement algorithms	Scheduling

Table IV.5: The family of bitmap counting algorithms: each algorithm is best suited for a different setting.

We used two configurations, one with a 12 second prefix and one with a 600 second prefix of the MAG+ trace. For each configuration we had 20 runs with the triggered bitmap algorithm, using different random hash functions. The average of the error for flows that had at least 4 connections was 13.6%.[10] Our algorithm reported 84.6% of the sources with 4 connections as reaching the threshold, 98.1% of those with 5, and all (100%) of the sources that had at least 8 connections. In Table IV.4 we report the *maximum* of triggered bitmap over the 20 runs. Triggered bitmap uses roughly 5 times less memory than Snort with the first configuration. For the more ambitious second configuration the gain increases to a factor of 9. With both configurations, triggered bitmap used less memory than probabilistic counting.

What do these results mean to a security analyst? Snort detects n connections with a maximum inter-event spacing of t. By default, Snort uses values such as n=4, t=3. Our technique uses significantly less memory at the expense of possibly missing

[10]This is an average over all sources. We noticed some "peculiarities": for sources that had 4 connections the average error was around 10.5% , for those with 5 around 11%, for those with 6 connections it was 18%, for those with 8 around 11.5% while for all others the averages were roughly in the range 14%-15.5%. We explain these as effects of having such a small direct bitmap.

port scanners. However, the probability of a port scanner not being detected decreases exponentially with the number of connections it opens. For example, the probability is 1.87% at 5 connections, 0.23% at 6, 0.03% at 7, etc. Using Snort's timing requirements, a fifth event must arrive within $t = 3$ seconds of the fourth event if the scan continues. Thus, we detect a continuing scan with probability 98.13% within 3 seconds and 99.77% within 6 seconds. Note also that port scans are usually the result of a brute-force network exploration such as Nmap [F98] or Code Red [M01]. Such tools frequently touch not just a handful of addresses, but an entire block of contiguous addresses. Thus, it is reasonable to expect a scan to continue after 4 events.

There are preliminary results on the use of triggered bitmap in an application computing per-IP source and destination statistics that is part of the CoralReef traffic analysis suite [KMKL+01]. The implementation and measurements reported here are the work of Ken Keys with contributions from David Moore, both from Caida. On a 10 minute OC-48 trace, the original application uses 316 megabytes of main memory. The improved version used a triggered bitmap with a 128 bit direct bitmap that allocated a multiresolution bitmap configured for an error of 5% after 4 bits were set. The memory usage decreased to 44 megabytes while the average error was 4.41%. The average running time was 349 seconds which is 29% below the running time of the original application (491 seconds).

Finally, note that because our algorithms reduce the memory usage by as much as an order of magnitude, they also enable detection of stealthy slow scans using the same amount of memory that naive algorithms use for fast scans. Because the memory required for each source is greatly reduced with our algorithms, we can afford to count more sources at a time. As a result, we can avoid timing-out state as aggressively as Snort and keep counting sources with longer inter-arrival times between events. By doing so, we can detect more stealthy port scans, a goal of many detection systems [SHM01].

IV.F Acknowledgments

This chapter is based on [EVF03] which is joint work with George Varghese and Mike Fisk. We thank Vern Paxson, David Moore, Philippe Flajolet, Marianne Durand,

Alex Snoeren and K. Claffy, Stefan Savage and Florin Baboescu for extremely valuable conversations.

IV.G Chapter summary

Using a suitably general definition of a flow, counting the number of active flows is important for a wide variety of security and networking applications such as detecting port scans and denial of service attacks, tracking worm populations, and calibrating caching. In this chapter we provide a family of bitmap algorithms solving the flow counting problem using extremely small amounts of memory. Most of the algorithms can be implemented at wire speeds (8 nsec per packet for OC-768) using SRAM since they access at most one memory location per packet, and can be implemented using simple hardware (CRC based hash functions, multipliers, and multiplexers). With the exception of direct and virtual bitmap, the algorithms are introduced for the first time in here.

The best known algorithm for counting distinct values is probabilistic counting. Our algorithms need less memory to produce results of the same accuracy. This can translate into a savings of scarce, fast memory (SRAM) for hardware implementations. It can also help systems that use cheaper DRAM to scale to larger instances of the problem.

In comparing head-on with probabilistic counting, our multiresolution algorithm works under the same assumptions and provides an error orders of magnitude lower when the number of flows is small and is slightly better for higher values. However, we believe our biggest contribution is as follows. By exposing the simple building blocks and analysis behind multiresolution counting, we have provided a family of *customizable* counting algorithms (Table IV.5) that application and hardware designers can use to reduce memory even further by exploiting application characteristics.

Thus, virtual bitmap is well-suited for triggers such as detecting DoS attacks, and uses 215 bytes to achieve an error of 2.773% compared to 2,076 bytes for probabilistic counting. Adaptive bitmap is suited to flow measurement applications and exploits stationarity to require 8 times less memory than probabilistic counting on sample traces.

Triggered bitmap is suited to running multiple instances of counting where many instances have small count values, which is the flow counting equivalent of identifying and measuring large traffic aggregates (Chapter III). For a port scan detection application, triggered bitmap used only 5.6 Mbytes on a 10 minute trace compared to the 49.6 Mbytes required by the naive algorithm and 22.3 Mbytes required by probabilistic counting. Using triggered bitmap resulted in a reduction by 29% in the running time and a factor of seven in the total memory usage of a traffic analysis application from the CoralReef suite. Applying flow sample and hold can further reduce memory costs when we are only interested in tracking sources with many connections. Given that low-memory counting appears to be useful in applications beyond networking which have different characteristics, we hope that the base algorithms in this chapter will be combined in other interesting ways in architecture, operating systems, and even databases.

Chapter V

Explicit, Concise Description of Traffic Mixes with Traffic Clusters

In Chapter III we presented identifying and measuring large traffic aggregates, an important building block of many traffic measurement systems. Various systems have different definitions for the aggregates they are interested in. For example, one can identify DoS victims by aggregating based on destination address, get an application breakdown by aggregating by source port and protocol and obtain a traffic matrix by aggregating by source and destination network. In this chapter we explore a method for explicitly describing the traffic mix through traffic clusters which generalize most of the useful ways of aggregating traffic.

Our new method of traffic characterization automatically groups traffic into minimal clusters of conspicuous consumption. It dynamically produces hybrid traffic descriptions that match the underlying usage. For example, rather than report hundreds of small flows, or the amount of TCP traffic to port 80, or the "top ten hosts", our method might reveal that a certain percent of traffic was used by TCP connections between AOL clients and a particular group of Web servers. Similarly, our technique can be used to automatically identify new traffic patterns, such as network worms or peer-to-peer applications, without knowing the structure of such traffic a priori. We describe a series of algorithms for constructing these traffic cluster reports and minimizing their representation.

V.A Introduction

The Internet is a moving target. Flash crowds, streaming media, content delivery networks, denial of service (DoS) attacks, network worms, peer-to-peer applications – these are but a few of the forces that shape traffic on today's networks. Each year, new applications and usage models emerge, and from these arise new communications patterns. This flexibility is a hallmark of the Internet architecture and can be credited with much of the Internet's success. At the same time, this quality also brings serious challenges for network management. Unlike the traditional voice networks, which are built around a single high-level abstraction for application data transfer ("calls"), managers of IP-based networks are forced to *infer* the type of traffic and how it relates to applications and users. Consequently, to understand and react to changes in network usage, a network manager must first analyze the bit patterns in individual packets, extract an appropriate traffic model, and then reconfigure network elements to recognize that model appropriately.

To make this process feasible in practice, managers use a standard set of predefined patterns to identify well-known aspects of network traffic. For example, network managers frequently construct a model of application usage by classifying traffic according to the TCP/IP header fields: Protocol, SrcPort and DstPort. Such an analysis might determine that 90 percent of traffic uses the TCP protocol, 75 percent of TCP traffic is for the HTTP service, 10 percent is for SMTP, 5 percent for FTP and so on. Similarly, to identify individual conversations between pairs of hosts, the five tuple (SrcIP, DstIP, Protocol, SrcPort, DstPort), is used to impart a "flow" abstraction on traffic. These kinds of analyses, exemplified by popular monitoring tools such as FlowScan, and Cisco's FlowAnalyzer, are a staple of modern network management [P00]. However, they have two significant limitations when used in practice: *insufficient dimensionality* and *excessive detail*.

While network traffic may be characterized by many different criteria, it is easiest to aggregate traffic along one dimension at a time. Unfortunately, by aggregating traffic along any single dimension, the network manager inevitably loses any interesting, but orthogonal, structure. For example, by aggregating traffic according to an

application-oriented view (i.e., Protocol and SrcPort), a network manager might conclude that peer-to-peer file sharing applications are in wide use, when in fact a small set of hosts are responsible for most of the file sharing traffic [SGDGL02]. While the network manager can expose this structure by using finer-grained representations, such as flows, she then must manage the excessive detail contained in such a representation. Rather than identifying the file-sharing traffic concisely, the flow-oriented view decomposes it into thousands of individual network transfers.

Consequently, network managers spend considerable time manually "hunting for needles" in their data – trying to understand what are the real and significant sources of traffic in their network and which components of usage are changing over time. This problem is only exacerbated when there is a pressing need to understand and respond to sudden traffic spikes such as network worms or denial-of-service attacks. Moreover, automated techniques for addressing these issues – such as *network pushback* [MBFI+01] – also require a means to identify malicious aggregates.

The focus of our work is to help automate these tasks, and this chapter makes four contributions in this direction. First, in Section V.B we motivate the need for a new approach to traffic monitoring that can automatically classify traffic into appropriate multi-dimensional clusters. We define a concrete and practical instance of such traffic clusters and describe a set of operations for reducing their size and increasing their utility. Based on this definition, Section V.C describes and evaluates algorithms for cluster construction and for implementing the key operations described in Section V.B. At the end of this chapter, we relate our efforts to previous work, discuss a proposal for future work, traffic synopses, and then summarize our results.

V.B Multi-dimensional Traffic Clusters

While the goal of every traffic analysis method is to empower the human operator with improved understanding, there is an inherent contradiction between the level of detail provided and the capacity of humans to absorb information: more detail can lead to a deeper understanding but makes the report harder to read – at one extreme is a bandwidth meter, at the other extreme are raw packet traces. The simplest solution

is to report only the largest flows, so-called "top ten" reports, but this approach has a serious flaw: aggregates made up of many small flows can be important, but each individual flow may not be large. For example, a busy Web server might generate the bulk of the traffic, but since all of its flows are relatively small, the "top ten" report might only contain transfers from a nearby FTP server hosting a few large files.

Another solution is to aggregate the individual flows into a common category (e.g. by source or destination port, source or destination address, prefix or Autonomous System number). However, if we chose the wrong dimension to aggregate over then we may miss the interesting characteristics of the traffic. For example, if we aggregate traffic by port number, we may miss the importance of traffic generated by a denial-of-service attack using random port numbers. In this case, aggregating traffic by destination address would likely be a more useful approach.

However, in some cases there is significant information that is hidden by aggregating on any *single* field, but is revealed by aggregating according to a combination of fields (multi-dimensional aggregates). For example, aggregating traffic by IP address might identify a set of popular servers and aggregating traffic by port might identify popular applications, but to identify which server generates which kind of traffic requires aggregating according to two fields simultaneously.

Our way out of this impasse is to focus on dynamically-defined traffic clusters instead of individual flows or other predefined aggregates. Our aim is to define the clusters so that any meaningful aggregate of individual flows is a traffic cluster. For example, a single cluster might represent all TCP client traffic originating from America Online's network destined for a cluster of replicated Web servers on Google's network. Another cluster might represent significant amounts of traffic originating from a host infected by the Sapphire worm destined to random addresses at UDP port 1434. While the goal is clearly attractive, automatically building such clusters is quite challenging. In practice, creating effective clusters requires balancing three key requirements:

- **Dimensionality**. The dimensionality of the problem is defined by how many distinct properties are considered in constructing a traffic cluster. If there is too little dimensionality then important traffic categories can be masked, while if there is too much, the computational overhead of computing cluster combinations can

become infeasible.

- **Detail**. While multi-dimensional clusters allow us to capture the structure of the traffic being analyzed, this does nothing to reduce the magnitude of data that must be evaluated – one can easily create thousands of clusters from a traffic trace. To make such data useful, a clustering algorithm must carefully prune this set to remove "unimportant" clusters and tradeoff the loss in detail for corresponding gains in conciseness.

- **Utility**. In the end, network managers are not merely passive observers of traffic, but are active parties who attempt to control and react to changes in traffic load and usage. Therefore, while we could construct traffic clusters using arbitrary combinations of packet header bit patterns, it is far more useful to restrict our choices to header fields that have clear semantic utility, are already well-classified by network hardware and can therefore be acted upon.

V.B.1 Defining Traffic Clusters

Based on these principles, we define our traffic clusters in terms of the five fields typically used to define a fine-grained flow: source IP address, destination IP address, protocol, source port and destination port. Unlike individual flows defined by unique values for each of these fields, clusters are defined by sets of values for each of these fields. These sets can contain a single value, all possible values (we use * to denote this case) or restricted subsets of possible values.

Evaluating all possible subsets of the values for each field would have made the problem of finding all large clusters intractable. Instead, we use the natural hierarchies that exist for each field. For IP addresses a cluster can be defined by prefixes of length from 8 to 32 (for individual IP addresses) or (*) for all IP addresses. For port numbers, clusters can be defined by a particular port number (e.g., port 80) or the set of all possible values (*). Because well known ports statically allocated for services are usually below 1024 and ephemeral ports allocated on-demand to clients are above 1023, the set of high (> 1023) port numbers and that of low (< 1024) ones can also define clusters. Finally, the protocol field can take on exact values or (*).

For example, the cluster defined as (SrcIP=10.8.200.3, DstIP=*, Proto=TCP, SrcPort=80, DstPort=*) represents Web traffic from the server with address 10.8.200.3. (SrcIP=*, DstIP=172.27.0.0/16, Proto=TCP, SrcPort=low, DstPort=high) represents TCP traffic coming from low ports and going to high ports destined to a certain prefix. Finally, (SrcIP=*, DstIP=*, Proto=ICMP, SrcPort=*, DstPort=*) represents all ICMP traffic. Notice that the first two clusters overlap with each other while the third cluster is disjoint.

There are many ways to further generalize the definition of traffic clusters. For example, we could define hierarchies based on Autonomous System number, integer ranges of port numbers, or arbitrary user-defined categories (e.g. Universities, Broadband Access, Data Centers, etc.). We could also employ heuristics such as those for identifying passive FTP and Napster traffic (that use random ports) in [P00]. While these additions may provide greater value in some settings, they do not require any fundamental changes in our approach, merely a different set of aggregation criteria when constructing clusters. For the remainder of this chapter we restrict our discussion to the "vanilla" cluster definitions we have described previously.

There are three advantages to this cluster definition. First, our definition is sufficiently general to capture much of the usage structure in existing applications and networks. Second, our definition is consistent with current packet classifiers [GM99a] and consequently a manager can apply controls, such as policy routing and hardware rate limiting, to the clusters we dynamically identify. Third, our definition allows a simple, visually appealing rule-based display of clusters (Section VII.A). Finally, initial results on several distinct networks (Section VII.B) indicate that clusters defined in this way do identify interesting resource consumption patterns that managers care about.

Our definition of clusters satisfies two of the requirements: the need for multi-dimensional clusters (dimensionality) and field selection constrained by existing field hierarchies (utility). However, satisfying the remaining requirement, reducing detail, requires significant additional effort.

V.B.2 Operations on Traffic Clusters

A *traffic report* is a list of clusters presented to a manager. It is very easy to see that even restricting ourselves to IP prefixes and very simple port ranges, that there is an exponential number of raw clusters. There are $2^{33} - 1$ possible source IP prefixes alone! The first step in reducing this onslaught of data is to restrict the report to only include *high volume* clusters, where volume may be defined as the number of bytes or the number of packets contained in the cluster over a predefined measurement interval. While other criteria could be used to filter clusters, data volume is a categorization of inherent interest. A cluster containing 20 percent of all traffic is one that a network manager is likely to care about, while a cluster that only contains a few packets usually warrants less attention.

However, even with this restriction, the number of such clusters identified in real traces is far too large to manage. Since the precious resource is not network bandwidth or CPU cycles, but a network manager's time, verbose and unstructured reports are not likely to be appreciated or useful. Consequently, to maximize the effectiveness of a traffic report, we believe that there are four essential operations that must be provided:

- **Operation 1, Compute:** Given a description of traffic as input (e.g., packet traces or NetFlow records), compute the identity of all clusters with a traffic volume above a certain threshold. This is the base operation.

- **Operation 2, Compress:** Having found the base set of clusters, one can compress the report considerably by removing a cluster C from the report if cluster C's traffic can be inferred (within some error tolerance) from that of cluster C' that is already in the report. For example, if all the traffic is generated by a single high-volume connection from source S to destination D is high volume, then one can infer that the total traffic sent by S is also high volume. Thus one should retain the S to D cluster for the detail it shows, and omit the S cluster as it can be inferred from the S to D cluster. Intuitively, the rule we use is to remove a more general cluster if its traffic volume can be inferred (within some error tolerance) from more specific clusters included in the report.

- **Operation 3, Compare:** A good way to save the manager time is to concisely

show how the traffic mix changes from day to day, or week to week (or on other timescales). Computing these deltas requires finding those high volume clusters that have changed significantly since the last report. This is trickier than it seems, because a high volume cluster on day 1 may now become low on day 2, or vice versa. Worse, more general clusters need not be larger than the sum of more specific non-overlapping clusters. Thus combining deltas (Operation 3) with compression (Operation 2) is much harder than just implementing each operation in isolation.

- **Operation 4, Prioritize:** Even after compressing the report and computing Deltas, it is still desirable to prioritize the elements of the report in terms of their potential level of interest to a manager. We choose to equate the interest in a cluster to what we call its *unexpectedness*. While there are many ways to define this metric, we chose to use a relatively unsophisticated approach that is easy to compute. We define unexpectedness in terms of deviation from a uniform model in which the contents of different fields is mutually independent. For example, if prefix A sends 25% of the traffic and prefix B receives 40% of all traffic, then under the assumption of independence, we would expect the traffic from A to B to be 25%*40%=10% of the total traffic. If the actual traffic from A to B is 15% of the total traffic instead of 10%, the cluster is tagged with a score of 150%, indicating that it is unexpectedly large by a factor of 1.5. If the traffic from A to B is only 6%, then it is given a score of 60%, indicating that it is unexpectedly small. The closer a score is to 100%, the more expected it is, and the less important it is to highlight to the user.

V.C Algorithms

The last section motivated four fairly abstract operations on sets of clusters. Here we describe the specific algorithms we chose to *implement* these operations. These algorithms form the engine that underlies the core of our AutoFocus tool described in Section VII.A. Rather than directly present the algorithms for the multi-dimensional case, we first present the simpler algorithms for the unidimensional (i.e., single field) case. Addressing this simpler case will help build intuition. Furthermore, AutoFocus

also includes in its output the simpler unidimensional results and the multidimensional algorithms use the results of the unidimensional algorithms to reduce their search space.

For some of the algorithms we also present theoretical upper bounds on the size of the report and on the algorithm's running time. Measurement results in Appendix G show that in reality reports are much smaller than these upper bounds. Since our focus in this chapter is maximizing information transfer to the manager, not algorithmic optimization; significantly faster algorithms that produce similar results may be possible.

In this section we use the terms dimension and field interchangeably, since each field defines a dimension along which we can classify. We use k for the number of fields. In the actual system we implemented, $k = 5$.

The sets (i.e., prefixes in IP address fields) for each field form a natural hierarchy in terms of set inclusion. This can be described by a tree where the parent is always the smallest superset of the child. The leaves of this tree are individual values the field can take. The root is always the set of all possible values, *. The sets denoted by two nodes are disjoint unless one of the nodes is an ancestor of the other, in which case it is a superset of the other. We call the number of levels in this tree the depth of the tree (the maximum distance from the root to a leaf plus 1). We use d_i for the depth of the hierarchy of the i-th of the k fields. The hierarchy for IP addresses we use in this chapter has a depth of 26 that for port numbers has a depth of 3 while for the protocol field we use the simplest possible hierarchy with a depth of $d = 2$.

The raw data we build our algorithms on is a simplified version of NetFlow flow records: each flow record, which we sometimes refer to as a "flow" for conciseness, has a key that specifies exact values for all five fields and two counters, one counting the packets that matched the key during the measurement interval considered and one for the number of bytes in those packets. Transforming a trace with packet headers and timestamps into such flow records leads to no loss of information from the standpoint of traffic clusters.

We use n for the number of such records. Each traffic cluster is made up of one or more flow records and the corresponding byte and packet counters are the sum of the corresponding counters of the flow records it includes. Note that if a cluster contains exact values in all fields it is exactly equivalent to a single flow. For the rest of this

section we ignore that the flow records contain two counters and work with a single counter. Our algorithms use a threshold H and focus on the traffic clusters that are above this threshold. We use s for the ratio between the total traffic T and the threshold $s = T/H$, so if H is 5% of the total traffic, then $s = 20$.

V.C.1 Unidimensional clustering

First we concentrate on the problem of computing high volume clusters on a single field, such as the source IP address. Note that even the unidimensional case is significantly more complex than traditional tools like FlowScan, in which managers define a static hierarchy by pre-specifying which subnets should be watched. By clustering automatically, we do not need to define subnets; the tool will automatically group addresses into "subnets" that contain a high volume of traffic.

We use d to represent the depth of the hierarchy and $m \leq n$ to represent the number of distinct values of the field in the n flow records of the input.

Computing Unidimensional Clusters

Before we describe our algorithms for computing the high volume unidimensional clusters, it is useful to bound their number. Consider the IP source address. A reasonable intuition might be that a threshold H of 5% of the total traffic restricts the report size to 20, because there can be at most 20 *disjoint* clusters, each contributing 5% of the traffic. Unfortunately, our definition of clusters allows clusters to overlap. Thus, if $128.50.*.*$ is a high volume cluster, then $128.*.*.*$ is as well. Fortunately, a given source address cluster's traffic can at most be counted in 25 other clusters (the number of ancestors in its hierarchy tree – we do not consider prefixes with lengths from 1 to 7). Therefore, the maximum number of high volume clusters is not 20 but roughly $20 \cdot 26 = 520$. This is formalized by:

Lemma 5 *The number of clusters above the threshold is at most* $1 + (d-1)s$.

Proof The counter of each flow record can contribute to at most $d-1$ sets besides *. The sum of the counters of all flows is T. The sum of the sizes of all clusters other than

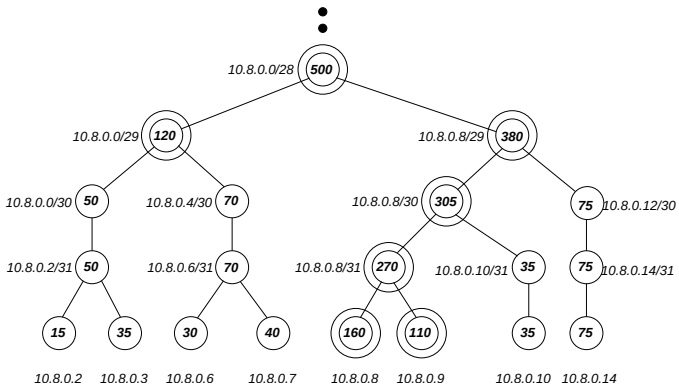


Figure V.1: Each individual IP address sending traffic appears as a leaf. The traffic of an internal node is the sum of the traffic of its children. Nodes whose traffic is above H=100 (double circles) are the high volume traffic clusters. The Web server 10.8.0.8 is a large cluster in itself. While no individual DHCP address is large enough, their aggregate 10.8.0.0/29 is, so it is listed as a large cluster.

* is at most $(d-1)T$. Therefore, there are at most $(d-1)T/H = (d-1)s$ clusters above H, and once we also add * we arrive at the final result. ■

For the unidimensional case, we now describe the algorithm to do **Operation 1**, computing the raw set of high volume clusters. When the number of sets in the hierarchy is relatively small, for example 257 for protocol and 65539 for port numbers, we can apply a brute force approach: keep a counter for each set and traverse all n flow records while updating all relevant counters; at the end, list the clusters whose counters have exceeded H.

If the number of possible values is much larger, as is the case for IP addresses, we use another algorithm illustrated by the example from Figure V.1. As we go through the flow records, we first build the leaf nodes that correspond to the IP addresses that actually appear in the trace. For example, there are only 8 possible source addresses (leaves) in the trace that Figure V.1 was built from. Thus, we make a pass over the trace updating the counters of all the leaf nodes. By the end of this pass, the leaf counters are correct; we also initialize the counters of all nodes between these leaves and the root to 0. In a second pass over this tree, we can determine which clusters are above threshold H by traversing in post order (children before parents). Also, just before finishing with

each node, the algorithm must add its traffic to the traffic of its parent. This way, when the algorithm gets to each node its counter will reflect its actual traffic. The memory requirement for this algorithm is $O(1 + m(d-1))$ and it can be reduced to $O(m+d)$ by generating the internal nodes only as we traverse the tree. The running time of the algorithm is $O(n+1+m(d-1))$. No algorithm can execute faster than $O(n+1+(d-1)s)$ because all algorithms need to at least read in the input and print out the result.

Compressing Unidimensional Traffic Reports

For the unidimensional case, we now describe the algorithm for **Operation 2**, compressing the raw set of high volume clusters. The complete list of all clusters above the threshold is too large and most often it contains redundant information. Even if a /8 (address prefix of length 8) contains exactly the same amount of traffic as a more specific /24 prefix, all the prefixes with lengths in between are also high volume clusters. More generally, perhaps an intermediate prefix length like /16 has a little more traffic than the /24 it includes (or the sum of the traffic of several more specific /24s already in the report) but not much more. Reporting the /16 adds little marginal value but takes up precious space in the report. Removing the /16 on the other hand, will mean that the manager's estimate of the /16 may be a little off. Thus we trade accuracy for reduced size. In general, define the compression threshold C as the amount by which a cluster can be off. In our experiments we defined $C = H$. We did so to avoid unintended errors: if the manager wants all clusters above H, surely she realizes that the report can be off by H in terms of missing clusters of size smaller than H. By setting $C = H$, we are only adding another way to be off by H. Also, setting $C = H$ produces the following simple but appealing result.

Lemma 6 *The number of clusters above the threshold in a non-redundant compressed report is at most s.*

Proof Since none of the clusters in the report is redundant, each has a traffic of at least $C = H$ that was not reported by any of its descendants. The sum of these differences is at most T because each flow can be associated with at most one most specific cluster in the report and these flows make up the difference between that cluster and the more

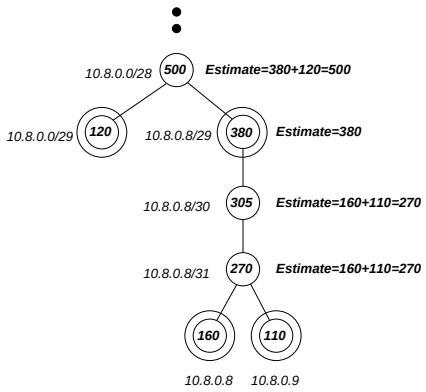


Figure V.2: The clusters from the compressed report are represented with double circles. Node 10.8.0.8/31 is not in the compressed report because its traffic is exactly the sum of the traffic of its children. Node 10.8.0.8/30 is not in the compressed report because its traffic is within a small amount (35) of as what we can compute based on its two descendants in the report.

specific ones. Therefore, a report can contain at most $T/H = s$ clusters. ■

If we go back to our original example for computing all clusters that send over 5% of the total traffic, we find that the number of clusters in the compressed report (assuming $C = H = 5\%$) is at most 20 and not roughly $20 \cdot 26$. The compressed report corresponding to Figure V.1 is in Figure V.2. Note that the number of nodes retained in the report (nodes with double circles) has dropped from 7 to 4 which is actually less than the 5 Lemma 6 would have bound the report size at for the 20% threshold.

Our algorithm for computing the unique non-redundant compressed report, exemplified in Figure V.2, relies on a single traversal of the tree of the high volume traffic clusters. Each node in the tree maintains two counters: one reflecting its traffic and one reflecting an estimate of its traffic based on the more specific clusters included in the report. We perform a post order traversal and decide for each node whether it goes into the report or not. We compute the node's "estimate" as the sum of the estimates of its children. If the difference between this value and the actual traffic is below the threshold, the node is ignored, otherwise it is reported with its exact traffic and its "estimate" counter is set to its actual traffic. This algorithm significantly reduces the size of the report while guaranteeing that all clusters of size H or larger can be

reconstructed within error C.

Computing Unidimensional Cluster Deltas

While compressed reports provide a complete traffic characterization for a given input, sometimes we are more interested in how the structure of traffic has *changed*. More specifically, the challenge is to produce a concise report that indicates the amount of the change for all the clusters whose increase or decrease in traffic is larger than a given threshold.

There are two ways to define the problem: by looking at the *absolute* change in the traffic of clusters, or by looking at the *relative* change. If the lengths of the measurement intervals are equal and the total traffic doesn't change much, one can use absolute change: the number of bytes or packets by which the clusters increase or decrease. However, to compare the traffic mix over intervals of different lengths (e.g. how does the traffic mix between 10 and 11 AM differ from the traffic mix of the whole day), we can only meaningfully measure relative change and must normalize the sizes of traffic clusters so that the normalized total traffic is the same in both intervals. Thus, even if the traffic of a given cluster changed significantly, if it represents the same percentage of the total traffic, its relative change is zero. For the rest of this chapter we assume that the we are computing the absolute change or that the traffic has already been normalized.

To detect the clusters that change by more than H, we can use the full traces from each interval, but a more efficient algorithm can be built simply using the uncompressed reports computed earlier. Since each cluster in the uncompressed report is above a threshold of H, if a cluster was below H in both intervals it could not have changed by more than H overall. But operating only on the reports for the two intervals still leaves some ambiguities: we cannot be sure whether a cluster that appears only in one of the intervals and is close to H was zero in the other one and thus changed by more than H, or was close to but below H and thus changed by very little. Of course, if the threshold used by the input reports is much below H, the ambiguity is reduced and we can ignore it in practice. A simple preprocessing step can provide the exact input required for the delta algorithm as follows: using reports with threshold H for both intervals we compute the set of clusters that were above H in either of them and in one more pass over the

trace we compute the exact traffic in both intervals for each of these clusters.

We can apply to delta reports a compression algorithm similar to that from the previous section. We decide whether to include a cluster into the compressed delta report by comparing its actual change to the estimate based on more specific clusters already reported: if the estimate is lower or larger by at least H than the actual change, the cluster is reported. Note that this can (and does) lead to putting clusters that did not change by more than the threshold into the compressed delta report. Consider the following example. The traffic from port 80 (Web) increased by more than the threshold and therefore we put it into the delta report. At the same time, the traffic from no other low port changed much and the total traffic from low ports remained the same. This is possible because traffic from many low ports may have decreased slightly, thus compensating for the increase in port 80 traffic. Our compressed delta report needs to indicate that the total traffic from low ports did not change because otherwise the manager would assume that it increased by approximately as much as the Web traffic.

Lemma 7 *The number of clusters in a non-redundant compressed delta report is at most* $s_1 + s_2$.

Proof Each cluster in the report covers traffic of at least H from one of the intervals that was not reported by any of its descendants. The sum of the absolute value of these differences is at most $T_1 + T_2$ (T_1 is the total traffic of the first measurement interval and T_2 is the total traffic of the second one) because each flow can be associated with at most one most specific cluster in the report and the sum of the sizes of all flows is $T_1 + T_2$. Therefore, there are at most $(T_1 + T_2)/H = s_1 + s_2$ clusters in the compressed delta report. ■

While this result suggests that compressed delta reports could be double the size of compressed reports, in practice traffic changes slowly, so the deltas are much more compact than compressed reports using the same threshold.

V.C.2 Multidimensional clustering

The relationships between multidimensional clusters form a more complex space defined by combining multiple unidimensional hierarchies. In the top of Figure V.3, the

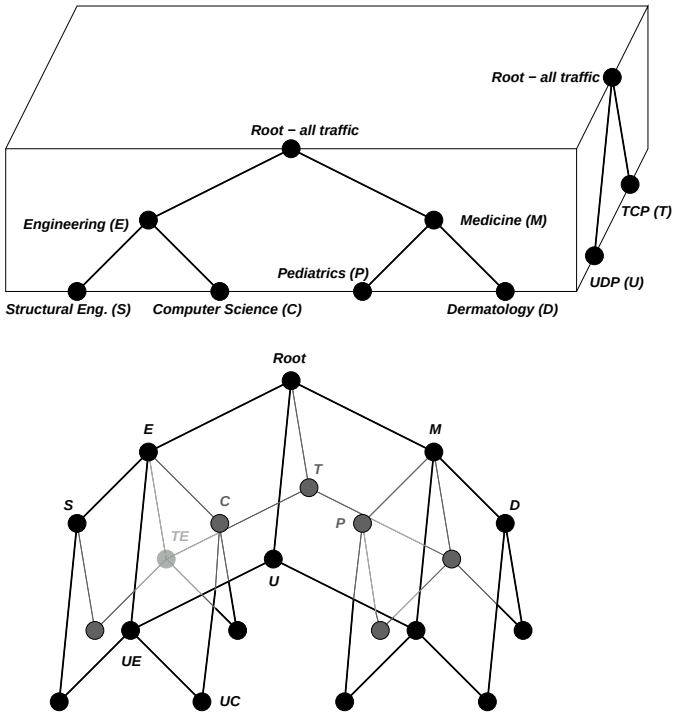


Figure V.3: The multidimensional model combines unidimensional hierarchies (trees) into a graph. The hierarchy on the near side of the cube breaks up the traffic by prefixes; the hierarchy on the right side of the cube by protocol. For example, the node labeled C on the near side represents the Computer Science Department, the node labeled U on the right side represents the UDP traffic and the node labeled UC in the graph represents the UDP traffic of the Computer Science Department.

closer face of the cube shows the prefix hierarchy that breaks up the traffic of a hypothetical university between the Engineering School and the Medical School, and breaks up the traffic of the Engineering School between the Structural Engineering Department and the Computer Science Department. On the right side of the cube we illustrate another hierarchy that breaks up the traffic by protocol into TCP and UDP. When we combine these hierarchies in the bottom of Figure V.3 we obtain a specific type of directed acyclic graph, a semi-lattice. Nodes in this graph have up to k parents instead of just one: one parent for each dimension along which they are not defined as *. For example, node UC represents the UDP traffic of the Computer Science Department and has as parents the nodes UE (the UDP traffic of the Engineering School) and C (the total traffic of the

Computer Science Department). Unlike unidimensional clusters, two multidimensional clusters can overlap and still neither includes the other: one can be more general along one dimension, while the second can be more general along another one. For example, clusters UE and C overlap (their intersection is UC) but neither includes the other. As a result, the size of the graph is much larger than the sizes of the trees representing the hierarchies of individual fields: it is the product of their sizes.

We use the phrase "unidimensional ancestor of cluster X along dimension i" to denote the cluster that is identical to X in its ith field and has wildcards in all the other $k-1$ fields. This is also a unidimensional cluster along dimension i. In our example, C is the unidimensional ancestor of UC along the prefix dimension and U is its unidimensional ancestor along the protocol dimension. We use the phrase "children of cluster X along dimension i" to denote the clusters that have exactly the same sets for all other dimensions and for dimension i their sets are one step more specific (i.e. they are children of the set used by X in the hierarchy of field i). For example S and C are the children of E along the prefix hierarchy and UE and TE are its children along the protocol hierarchy.

Computing Multidimensional Clusters

Our algorithm examines all clusters that may be above the threshold; for each such cluster, the algorithm examines all n flows, and adds up the ones that match. If the traffic is above the threshold, the cluster is reported, otherwise it is not. Explicitly evaluating all the clusters generated by the n flows in the input, approximately $n \prod_{i=1}^{k} d_i$, is not an acceptable approach for the configurations we ran on. Therefore our algorithm restricts its search (thereby reducing running time) based on a number of optimizations that prune the search space.

The first optimization exploits that all the unidimensional ancestors of a certain cluster include it, so the cluster can be above the threshold only if all its unidimensional ancestors are also above threshold. We first solve the k unidimensional problems. After this, we restrict the search to those clusters that have field values appearing in each of the uncompressed unidimensional reports. Next, observe that traversal of the search space is such that we always visit all the ancestors of a given node before visiting the node

itself. Thus our second optimization is to consider only clusters with all parents above the threshold. This is very easy to check because in our graph nodes have pointers to their parents. A third optimization is to batch a number of clusters when we go through the list of flow records.

Even with all three optimizations, among all our algorithms, this one produces the largest outputs and takes the longest to run. For example, computing both packet and byte reports with a 5% threshold takes on average 16 minutes for a one day measurement interval, 2 minutes for a one hour measurement interval and 1 minute for a five minute measurement interval using a 1 GHz Intel processor, however using a threshold of 0.5% for a one day trace it takes over 3 hours to compute the uncompressed report. We believe this algorithm can be improved significantly. Lemma 8 bounds the number of high volume clusters in the multidimensional case. While there are pathological inputs that could force the size of the output close to its worst case bound, the results for real data are much smaller.

Lemma 8 *The number of clusters above the threshold is at most* $s \prod_{i=1}^{k} d_i$.

Proof The counter of each flow can contribute to at most d_i sets along dimension i (there can be less than d_i because some leaves of the tree might be closer to the root). There are at most $\prod_{i=1}^{k} d_i$ ways of combining these sets into clusters so each flow contributes to at most $\prod_{i=1}^{k} d_i$ clusters. The sum of the counters of all flows is T. The sum of the sizes of all clusters is at most $T \prod_{i=1}^{k} d_i$. Therefore, there are at most $T/H \prod_{i=1}^{k} d_i = s \prod_{i=1}^{k} d_i$ clusters larger than H. ■

Compressing Multidimensional Traffic Reports

For the multidimensional case, **Operation 2**, compression, is absolutely necessary to achieve reports of reasonable size. We first bound the maximum size of the compressed report.

Lemma 9 *For any traffic mix, there exists a multidimensional compressed report of size at most* $(s \prod_{i=1}^{k} d_i)/(max\ d_i)$.

Proof Let m be the field with the deepest hierarchy ($d_m = max\ d_i$). Let L_j be the sizes of clusters (indexed by j) that have * in field m. Since each flow belongs to at most

```
COMPRESS_REPORT
1   sort_more_specific_first(cluster_list)
2   foreach cluster in cluster_list
3       for field = 1 to 5
4           sum[i]=add_estimates(cluster.childlists[field])
5       endfor
6       cluster.estimate = max(sum[i])
7       if(cluster.traffic − cluster.estimate ≥ H)
8           add_to_compressed_report(cluster)
9           cluster.estimate = cluster.traffic
10      endif
11  endforeach
```

Figure V.4: The algorithm for compressing traffic reports traverses all clusters starting with the more specific ones. The "estimate" counter of each cluster contains the total traffic of a set of non-overlapping more specific clusters that are in the compressed report. The clusters whose estimate is below their actual traffic by more than the threshold H , are included into the compressed report.

$\prod_{i \neq m} d_i$ clusters with $*$ in field m, we get $\sum L_j \leq T \prod_{i \neq m} d_i$. We can obtain any cluster by varying the mth field of the corresponding cluster j. We can compress all the clusters obtained from cluster j by varying field m using the unidimensional algorithm for field m, so by applying Lemma 6, we get that the number of clusters in the result is bound by $s_j = L_j/H$. These reports for all j together cover all clusters, so for the total size of the report we get $\sum L_j/H \leq s \prod_{i \neq m} d_i = (s \prod_{i=1}^{k} d_i)/(max\ d_i)$. ■

We have implemented a fast greedy algorithm for multidimensional compression (Figure V.4). It traverses all clusters in an order that ensures that more specific clusters come before all of their ancestors (line 1). At each cluster we keep an "estimate" counter. When we get to a particular cluster we compute the sum of the estimates of its children along all dimensions (line 4) and set the estimate of the current cluster to the largest

among these sums (line 6). If the difference between the estimate and the actual traffic of the cluster is below the threshold (line 7), it doesn't go into the compressed report. Otherwise we report the cluster (line 8) and set its "estimate" counter to its actual traffic (line 9). The invariant that ensures the correctness of this algorithm is that after a cluster has been visited, its "estimate" counter contains the total traffic of a set of non-overlapping more specific clusters that are in the compressed report. It is easy to see how this invariant is maintained: when computing the estimate for the cluster, for each dimension, the algorithm computes the sum of the estimates of the children of the node (cluster) along that particular dimension. Since the sets at the same level of the field hierarchy never overlap, the sets contributing to the estimates of distinct children will never overlap, so the invariant is maintained.

The compression rule allows the algorithm to consider all non-overlapping sets of more specific clusters reported when computing the estimate. Our algorithm does something simpler: it only looks at the sets of non-overlapping more specific clusters that can be partitioned along a dimension or another. Thus *it will sometimes include clusters into the report that could have been omitted, but it will never omit a cluster that does not meet the compression criterion* (i.e., is larger by more than H than the traffic of each of the sets of non-overlapping more specific clusters in the compressed report). This is a small price to pay for the big gains in performance we get by performing simpler local checks. Computing both byte and packet compressed reports takes less than 30 seconds for a threshold as low as 0.5%.

In practice compressed traffic reports are two to three orders of magnitude smaller than uncompressed reports and dramatically smaller than the theoretical bound. For a threshold of 5% of the total traffic, the average report size is around 30 clusters. This is not influenced significantly by the length of the measurement interval or the diversity of the traffic (backbone versus edge), but the size of the report is proportional to the inverse of the threshold. The results of our measurements are presented in Appendix G

Computing Multidimensional Cluster Deltas

While in the unidimensional case the computation of delta reports was a simple extension of the compression algorithm, in the multidimensional case the interactions

between compression (**Operation 2**) and deltas (**Operation 3**) are more complex.

Our algorithm takes as input uncompressed reports for the two intervals. We assume that these reports contain the exact traffic of all clusters that are above H in either of the intervals (see Section V.C.1 for a discussion about the use of other types of input such as full traces). Based on these two reports we build the graph representing the relationships between clusters above the threshold in any of the intervals. For all nodes we compute the change in traffic – the explicit list of the clusters that changed (increased or decreased) by more than H. This could be considered our uncompressed delta report. For compressing the delta report we use a greedy algorithm similar to our multidimensional algorithm: we traverse the clusters, more specific first, and for each node, based on the more specific clusters we already decided to add to the delta report, we decide whether the current cluster needs to be added or not. However, the procedure we use for deciding if a cluster is added or not is quite different.

When computing the compressed report in the previous section, we were looking for the set of non-overlapping clusters (each more specific than the current cluster and each in the compressed report) that had the largest traffic. The situation is different for the delta report. One thing that doesn't change is that if a set of more specific clusters includes another set we ignore the subset. Here by one set of clusters including another we mean that it matches all flows the other set matches; it can be the case that the second set has clusters that are not in the first set, but they are more specific than some clusters from the first set. We call a set of non-overlapping clusters *maximal* if there is no other set of non-overlapping clusters that includes it. Change can be negative or positive, so if we have two maximal sets we cannot ignore the one with a smaller change (as we could in the case of simple compression).

Our rule for deciding whether to leave out a cluster from the delta report is to ensure *all* maximal sets of more specific clusters are within the threshold (either direction) of the actual change for the cluster. Because we need to look at all maximal sets of more specific clusters we cannot use the simplification used by the earlier report compression algorithm that made all decisions locally by restricting itself to sets that can be partitioned along one of the dimensions. Despite a lack of polynomial bounds for the running time of this algorithm, our implementation made decisions on clusters with

up to 50 more specific cluster in the change report within seconds.

Computing "Unexpectedness"

Recall **Operation 4**, which seeks to prioritize clusters via a measure of unexpectedness based on comparing the cluster percentage to the product of the percentages computed for each field in the cluster by itself. Computing the unexpectedness score of a given cluster is very easy using the graph describing the relations between the high volume clusters: we only need to locate the unidimensional ancestors along all dimensions.

V.D Related work

The problems we are solving are related to classical clustering [HTF01], but are different in that we use the space defined by the field hierarchies instead of a Euclidian space. Another problem, finding association rules [AIS93], requires finding frequent item sets in high dimensional data, and is a well studied problem in data mining. The two important differences between these two problems are: *1)* Most approaches to association rules do not use hierarchies. A notable exception is Han and Fu [HF95] who use a *single* hierarchy across all fields unlike our use of *separate* hierarchies for each field. *2)* Our compression rules were crucial to the effectiveness of AutoFocus. To the best of our knowledge, no algorithms for association rules use compression rules similar to ours.

V.E Future work: traffic synopses

In Chapter III we presented identifying and measuring large traffic aggregates, an important building block of many traffic measurement systems. Various systems have different definitions for the aggregates they are interested in. For example, one can identify DoS victims by aggregating based on destination address, get an application breakdown by aggregating by source port and protocol, and obtain a traffic matrix by aggregating by source and destination network. We need separate instances of the building block for identifying large aggregates for each of these measurement tasks the system might be used for. What we would really like is an initial building block that can provide raw data to be shared by all these systems to accurately compute their respective

aggregates. As we saw in Chapter IV, often the traffic is measured in flows, not bytes or packets, and the algorithms to count flows are quite different from those used to count bytes or packets. An initial building block to be shared by all systems must be able to support flow counting applications too. Both of these building blocks can be efficiently implemented at high speed but they are specialized, unlike the traffic clusters discussed in this chapter.

In this section we define a synopsis of the traffic mix as a form of raw data to be generated by an early building block that allows wide flexibility in the aggregation of the traffic and provides accurate estimates of those aggregates. The main focus of this section is defining the problem addressed by network traffic synopses, which we do in Section V.E.2. We also propose solutions to the problem, but without detailed analyses and measurements. We discuss two synopses for systems that measure the traffic in packets in Section V.E.3. In Section V.E.4 we adapt them to count bytes. In Section V.E.5 we present a different synopsis for systems that measure the traffic in flows.

V.E.1 Introduction

The idea of synopsis data structures has been explored by Gibbons and Matias [GM99] in the context of massive databases where keeping a small summary of the data helps give quick approximate results to many important queries. Sketches [GKMS01] based on properties of certain distributions of random variables have also been proposed for achieving similar purpose. Some of the synopsis algorithms we propose in this section build on those proposed in the database context. But because the requirements are different (e.g. low worst case per packet processing as opposed to low average processing), some database solutions are not directly applicable.

V.E.2 Problem definition

A network traffic synopsis is a compact summary of a traffic mix with respect to some measure such as the number of packets, bytes or flows. It is defined through the following five properties

- **Flexibility of aggregation**: Analyses using the traffic synopsis should be able to group the traffic into aggregates based on any function of the fields (attributes) captured by the synopsis. For example, if source and destination IP address are among the fields of the synopsis, one should be able to aggregate the traffic by source IP, by source and destination IP pair, or by destination prefix or autonomous system;

- **Accuracy**: Based on the synopsis we can estimate without bias the traffic of any valid aggregate exceeding a certain percentage of the total traffic, and with high probability the estimate is close to the actual traffic of the aggregate. For example we can expect to be able to estimate the traffic of all aggregates above 1% of the total traffic with a standard error of no more than 10%;

- **Compactness**: The synopsis data structure uses a small, fixed amount of memory that depends on the accuracy desired but not on the traffic volume and traffic mix;

- **Additivity**: We should be able to combine similar synopses into synopses with the same size and accuracy that describe the combination of the traffic mixes. For example, if we have synopses describing the traffic of each day of the week, we should be able to combine them into a synopsis that describes the traffic of the whole week, or if we have synopses for the traffic of all links between two POPs we should be able to combine them into a synopsis of the total traffic between the two POPs;

- **Easy computation**: There should be algorithms for computing the synopsis in a streaming fashion, looking at packets as they arrive, using memory not significantly larger than the size of the synopsis data structure, and with little per-packet processing. Having a low *worst case* packet processing is preferable to only having a low *average* packet processing because it makes hardware implementation simpler by avoiding reliance on packet buffers.

V.E.3 Packet synopses

In this section we address the problem of packet synopses: network traffic synopses for systems that measure the traffic in packets. We discuss two different solutions.

Reservoir synopses have the advantage of being very simple and easy to implement. Counting synopses are more accurate, but they are also more complex, and whether they can be implemented with low worst case per-packet processing bounds is an open question.

Reservoir synopsis

The simplest and easiest way to implement a synopsis is a sampling stream of packets. This is what sFlow [PPM01] delivers to the traffic analysis application and also what the flow cache of sampled NetFlow [CSN] sees. While based on randomly sampled packets we can get an unbiased estimate of the number of packets (or bytes) in any aggregate, we also want to have an estimate that has low variance.

But the size of the sample grows as the product of the sampling rate, and the number of packets in the traffic mix which is not known in advance. Reservoir sampling [V85] avoids the problem of having to choose a sampling rate in advance by always keeping a fixed size reservoir of samples. It replaces old entries with new ones in a way that ensures that the current reservoir is equivalent to a sample of the traffic seen so far generated at an appropriate sampling probability. To compute estimates of traffic aggregates we only need to count the number of matching packets in the reservoir and multiply by the inverse of the current sampling rate.

One can maintaining the reservoir synopsis very efficiently. While the reservoir is not full, all packets are added. After that, the random decisions at each packet about whether to add it and if so which sample to replace can precomputed offline based on the appropriate probability distributions. The reservoir itself can be a simple table with the relevant header fields of the sampled packets. If the speed of the memory does not allow one update per packet and the number of packets is much larger than the size of the reservoir, we can randomly pre-sample the traffic and do no harm with a probability that would allow the memory to keep up without losing any of the statistical properties of the reservoir.

Combining two reservoir synopses is straightforward. We divide the entries in the resulting synopsis among the two input synopses proportionally to their total traffic and select randomly the elements from the input synopses to occupy those entries. This

procedure can be easily generalized to more than two input synopses.

Counting synopsis

We saw in Section III.F that sample and hold and multistage filters can provide more accurate results than random sampling. Are there algorithms that can provide better accuracy than sampling and yet allow flexible aggregation?

A clear confirmation that this is possible comes from flow sampling [DLT01], which uses independent random sampling of flow records to reduce the amount of data used by a billing application to compute estimates of customer traffic. By making the sampling probability depend on the flow sizes in a way that favors large flows, flow sampling improves the accuracy of the estimates for the aggregates computed later on. But flow sampling uses as input the whole set of flows, which is not feasible to compute in early stages of the measurement system due to memory limitations.

Gibbons and Matias propose [GM98] a synopsis data structure called counting samples that uses fixed amounts of memory to keep per flow counters for flows based on packets selected at random from the data stream. This algorithm is closely related to sample and hold (Section III.D.1). We can properly account for the number of packets that went by before the first packet got sampled by adding the inverse of the sampling rate. Based on these counts we can obtain an unbiased estimate of any traffic aggregate. The accuracy improvements with respect to reservoir synopses can be significant, but they depend on the aggregation performed in the later analysis stages. If all flows have one packet or very few of them, reservoir synopsis and counting synopsis are equivalent.[1] If a considerable fraction of the traffic is flows with more packets than the inverse of the packet sampling rate, the results provided by counting samples are always more accurate, even if the aggregate we are looking at has only one packet flows. The reason for this is that the counting synopsis can afford to sample more often. If the aggregate consists of few large flows, the accuracy benefits of the counting synopsis are even larger. In the extreme, the aggregate consists of a single large flow and then the analysis from

[1]This holds under the assumption that the memory cost of an entry in the counting synopsis is the same as for one in the reservoir synopsis. Since the counting synopsis needs space for the counter and has some overhead associated with each entry because it needs to do lookups on incoming packets, this is not true. So if all flows had a single packet, a reservoir synopsis would actually be better than a counting synopsis.

Section III.F applies and the accuracy benefit is maximal.

Sample and hold has the dilemma of selecting the right sampling rate to fill the available memory with entries. Counting samples solve this problem in a similar manner to reservoir sampling: they make sure that at all times the memory is close to full and that it represents a valid synopsis of the traffic. The way counting samples achieve this is by changing the sampling rate and reprocessing all the entries (and probably dropping some) whenever the table is full. The average per-packet processing can be still kept low, but the worst case involves an expensive traversal of the whole memory. How to achieve similar effects with low worst case per-packet processing bounds is an open question that should be answered before counting synopses can be deployed close to the wire. Until then, they seem like a good choice for aggregation points that process in software the incoming raw measurement data. We also note here that the per-packet processing involves a flow lookup which translates to extra processing overhead if implemented in software, or extra transistors if implemented with content addressable memories in hardware.

Converting between reservoir synopses (more likely at the early stages of the measurement system) and counting synopses (more likely at the late stages of the measurement system) is a useful operation. How to combine counting synopses to achieve the additivity property is an open problem.

V.E.4 Byte synopses

Measuring the traffic in packets and bytes are related problems. However, due to the existing differences we must address byte synopses separately.

Reservoir synopsis

We discuss here two ways of adapting the packet reservoir synopsis to a byte reservoir synopsis: adding packet sizes and changing the selection rule for samples to take into account sample sizes.

Including packet sizes into the samples stored by the reservoir allows us to compute unbiased estimates of the size of aggregates in bytes by multiplying the size of selected packets with the inverse of the packet sampling rate. The variance of the esti-

mates will depend on the size of the packets that make up the aggregate: aggregates with larger sized packets will have a larger variance. As long as the ratio between maximum and minimum packet sizes do not exceeds the current value of 40, this difference among the variances might be less important than the advantage of simplicity and the savings we can obtain by sharing the same traffic synopsis for both packet and byte reports.

Another alternative is to take into account packet sizes when selecting the samples, thus selecting packets with probability that depends on their size. How exactly the probability should depend on packet size is an open question. Another open problem is the initialization of the synopsis (how to add samples before the synopsis fills up).

Counting synopsis

Packet counting synopses can be easily converted into byte counting synopses by changing the probability of a packet creating an entry and making it dependent on packet size. This is equivalent to conceptually sampling the bytes within the packet, not the packet itself. To achieve unbiasedness one can even ignore part (roughly half) of the packet, since if a byte inside the packet is selected, the ones in the packet preceding it should not be counted.

A simple solution is to keep byte and packet counting synopses separate. While there might be common entries with both many bytes and many packets, the overlap is not perfect and combining the two into a single structure is not a straightforward operation.

V.E.5 Flow synopsis

Random sampling of the packet stream provides a basis for unbiased estimation for the byte and packet counts of arbitrary aggregates based on packet header fields. Using more elaborate techniques that extend the idea of random sampling improved the accuracy of the synopsis. It is natural to ask whether we can also start from a random sample of the stream if we count traffic as flows. Unfortunately, no. It has been shown that for any estimator that computes an estimate of the number of flows based on a random sampling of the packets belonging to a source IP, there is a traffic mix for which the estimate is far off from the actual count [CMN98]. For example, if

we multiply the number of flows we count in the sampled packets by the inverse of the sampling probability, we get accurate flow counts if the flows contain a single packet, but we severely overestimate if all flows have many packets (and thus they all have at least one packet pass sampling). If we do not adjust the flow count, we underestimate when the traffic mix has short flows. Using additional information such as TCP SYN flags in the sampled packet headers can lead to more robust estimates [DLT03]. These methods apply only to TCP and rely on the endhosts correctly setting the header flags. On the other hand, our bitmap algorithms from Chapter IV can give unbiased and accurate estimates for the number of flows using small amounts of memory, so we will look at them for inspiration.

As a starting point for the development of a synopsis for flow counts, we use a flow counting algorithm called adaptive sampling, proposed by Wegman and described by Flajolet [F90]. The algorithm keeps a table with all flow identifiers seen. When the table is filled, it retains only a sample of the table using a hash function on the flow identifier in the same way as virtual bitmap (Section IV.C.2) to sample the flow ID space. While this algorithm is simple and the table is a good synopsis for applications that measure traffic in flows, its worst case per-packet processing is high (a traversal of the entire table) and thus it is not directly suitable for high speed network traffic measurement.

The flow synopsis we propose would have a fixed size flow ID table that would be initially filled with the flow ID seen in the traffic. For the correct operation of the algorithm we also need to compute a hash of the flow ID for each packet. The hash should be based on a secret seed to avoid the possibility of attacks against the synopsis. The value of the hash is used to decide which flow IDs to keep: we always fill the synopsis with the flow IDs with the highest values seen so far. Based on order statistics, we can estimate the number of flows in each aggregate based on the number of flow IDs belonging to the aggregate that are part of the synopsis and/or the lowest hash value of these flow IDs. A study of the variances of the two methods will allow us to chose the best one.

The flow lookup required by a flow synopsis can be easily implemented in hardware using CAMs. We can use systolic arrays [L79, LS94] or other hardware solu-

tions [MSR97] to keep track of the entry whose flow ID has the lowest hash.

To combine multiple flow synopses they must have used the same hash for selecting the entries to keep. If so, we only need to perform the set union of the two flow synopses and keep those with the highest hashes for their flow IDs.

Let us conclude this section with an example that illustrates the power of the first property of traffic synopses: flexible aggregation. Assume that we are looking for scans of our network. A flow synopsis from the edge of the network can be used to identify all sources with many flows. But what we are looking for is not exactly sources with many flows, but source IPs that connect to many destination IP, destination port pairs. For example not only a scanner has many flows but also an IP that repeatedly accesses the same service on the same server using many random client ports. Can we tell the two apart? Yes, because the flow IDs associated with the second one will clearly identify the server and the port it keeps connecting to, whereas the flow IDs for the scanner will show how it covers many destination IP, destination port pairs.

V.E.6 Summary

In this section we defined network traffic synopses. We differentiated between three types of synopses based on how they measure the traffic: bytes, packets or flows. Traffic synopses are summaries of the traffic mix defined by five important properties: flexibility of aggregation, accuracy, compactness, additivity and easy computation. We discussed two synopses suitable for measuring the traffic in packets and bytes: reservoir synopsis and counting synopsis. Our flow synopsis is suitable for systems that measure the traffic in flows. Future work on these synopses is necessary to ensure that they satisfy all the requirements imposed by the definition of traffic synopses and to evaluate their accuracy on real work traffic mixes.

V.F Acknowledgments

This chapter is based on [ESV03] which is joint work with George Varghese and Stefan Savage. We would like to thank Vern Paxson and Jennifer Rexford for the many discussions that led to the clarifying the concept of traffic clusters. We would also

like to thank Ranjita Bhagavan for discussions about the feasibility of certain hardware solutions for synopses.

V.G Chapter summary

Managing IP-based networks is hard. It is particularly complicated by not understanding the nature of the applications and usage patterns driving traffic growth. In this chapter, we have introduced a new method for analyzing IP-based traffic, *multidimensional traffic clustering*, that is designed to provide better insight into these factors. Traffic clusters generalize dimensions of aggregation commonly used in traffic measurement. Our traffic reports describe the large aggregates in the traffic mix in an explicit way that makes them readable to humans.

The novelty of our approach is that it automatically infers, based on the actual traffic, a traffic model that matches the dominant modes of usage. Unlike previous work, our algorithms can analyze traffic along multiple different dimensions (Source address, Destination address, Protocol, Source port, Destination Port) at once, at the right granularity. Our novel compression rule makes traffic reports very concise without any significant loss in the accuracy of the description. Our compressed delta reports address the related problem of describing the difference between two traffic mixes and can help spotting changes in the traffic mix.

Chapter VI

Adapting Traffic Measurement to the Traffic Mix

Chapters III and IV present algorithmic solutions for various traffic measurement building blocks, together with their analyses which give strong upper bounds on their memory usage, processing time and output size. We can use these analyses to configure the algorithms in a way that guarantees that they fit within the limitations (Figure I.1) of the traffic measurement system. Conservative dimensioning based on worst case bounds guarantees that the algorithms work with all traffic mixes. However, normal traffic mixes are often more favorable and leave most resources of a conservatively dimensioned device unused. A more aggressive choice of parameters can lead to better resource usage and more accurate results.

This chapter explores methods for adapting the configuration of our algorithmic building blocks to the traffic mix. Our algorithms work on measurement intervals. There are two distinct ways of doing adaptation. Inter-interval adaptation discussed in Section VI.A uses information about the traffic mix in the previous intervals to set configuration parameters for the current interval. Intra-interval adaptation discussed in Section VI.B adjusts the configuration parameters within the interval based on the observed traffic.

ADAPTTHRESHOLD

$usage = entries_used/flow_mem_size$

if $(usage > target)$

$threshold = threshold * (usage/target)^{adjustup}$

else

if (threshold did not increase for 3 intervals)

$threshold = threshold * (usage/target)^{adjustdown}$

endif

endif

Figure VI.1: Dynamic threshold adaptation to achieve target memory usage

VI.A Inter-interval adaptation

Inter-interval adaptation relies on the fact that most often the traffic mix does not change significantly from one measurement interval to the next one. This is especially so when the measurement intervals are short, say 5 seconds.

VI.A.1 Adapting the threshold for algorithms for identifying large flows

The measurements from Section III.H show that using more aggressive configurations than those based on the theoretical analysis from Sections III.E.1 and III.E.2 can lead to substantial gains. For example the number of false positives for a multistage filter can be four orders of magnitude below what the conservative analysis predicts. To avoid a priori knowledge of flow distributions, we adapt algorithm parameters to actual traffic. The main idea is to *keep decreasing the threshold below the conservative estimate until the flow memory is nearly full* (totally filling the memory can result in new large flows not being tracked).

Dynamically adapting the threshold is an effective way to control memory usage. Sampled NetFlow uses a fixed sampling rate that is either so low that a small

percentage of the memory is used all or most of the time, or so high that the memory is filled and NetFlow is forced to expire entries which might lead to inaccurate results exactly when they are most important: when the traffic surges.

Figure VI.1 presents our threshold adaptation algorithm. There are two important constants that adapt the threshold to the traffic: the "target usage" (variable *target* in Figure VI.1) that tells it how full the memory can be without risking filling it up completely, and the "adjustment ratio" (variables *adjustup* and *adjustdown* in Figure VI.1) that the algorithm uses to decide how much to adjust the threshold to achieve a desired increase or decrease in flow memory usage. To give stability to the traffic measurement device, the $entries_used$ variable does not contain the number of entries used over the last measurement interval, but an average of the last 3 intervals.

We use measurements (presented in Section VI.A.2) to find good values for the target usage and the adjustment ratio. We want the target usage as high as possible, but still low enough so that the short-term fluctuations in the number of large flows do not cause the flow memory to fill up. Based on measurements, we use a target memory usage of 90% in our experiments. The adjustment ratio reflects how our traffic measurement device adapts to longer term fluctuations in the number of large flows. When the memory is above the target usage, we are drastic in increasing the threshold, but when the usage is below the target we are cautious in decreasing it. By measuring the highest and lowest impact the increase of threshold has on the number of flows in the flow memory, we arrived to a value of 3 for *adjustup*, 1 for *adjustdown* in the case of sample and hold and 0.5 for multistage filters.

VI.A.2 Calibrating the threshold adaptation algorithm

In this section we use measurements to determine the right constants to be used by the algorithm for dynamically adapting the threshold. We will determine different parameters for sample and hold and multistage filters. We first determine the safety margin and then the range of adjustment ratios.

Trace (+flow ID)	Sample and hold		
	o=1	o=4	o=7
MAG	78.0%/92.8%	87.2%/94.4%	91.0%/95.0%
MAG destIP	73.6%/93.1%	88.6%/94.8%	90.2%/95.7%
MAG ASpair	82.3%/92.1%	87.1%/93.0%	87.8%/93.7%
IND	78.0%/92.5%	88.8%/94.2%	87.9%/94.4%
COS	83.9%/90.0%	85.7%/90.7%	86.6%/91.6%
	Multistage filters		
	d=2	d=3	d=4
MAG	72.6%/91.3%	76.4%/92.1%	81.5%/93.0%
MAG destIP	65.1%/92.8%	65.7%/94.3%	85.5%/94.7%
MAG ASpair	63.9%/92.1%	69.5%/93.4%	70.0%/93.8%
IND	75.5%/91.7%	67.0%/92.4%	32.0%/92.0%
COS	72.1%/89.0%	66.7%/89.2%	52.1%/89.2%

Table VI.1: The average to maximum memory usage ratios for various configurations

Finding the right target usage

We use a brute force approach to finding the right measurement interval: we run the algorithms with a large number of configurations and thresholds on all traces and with all flow definitions and record the ratio between the average and maximum memory usage for each configuration. The results in table Table VI.1 show the minimum and average values (over all configurations). We tested thresholds between 0.005% and 1% of the link bandwidths in increments of around 40%. For sample and hold we preserved entries, used an early removal threshold of 15% and used oversampling of 1, 4 and 7. For multistage filters we used parallel filters with conservative update, preserving entries and shielding. The number of counters goes from less than the number of new large flows per interval for the smallest threshold up to 8 to 64 times more in increments of a factor of 2 (4 to 7 configurations) and for each number of counters we measure filters with depths of 2, 3 and 4 stages. To avoid pathological cases we do not consider the configurations where the average number of memory locations used is less than 100. We can see that for all algorithms and all traces the average ratio between the average and maximum memory usage is between 89% and 96%, but the worst case numbers are much smaller. Furthermore these numbers do not depend significantly on the number of stages or oversampling. We can also see that the minimum ratios are smaller for

Trace (+flow ID)	Perfect knowledge	Sample and hold		
		o=1	o=4	o=7
MAG	0.34/1.48	1.00/1.78	1.18/1.98	1.25/2.13
MAG destIP	0.45/2.86	1.00/2.78	1.21/2.97	1.31/3.06
MAG ASpair	0.80/3.30	1.09/3.40	1.38/3.63	1.56/3.81
IND	0.95/2.27	1.23/2.97	1.38/3.64	1.35/3.76
COS	0.77/3.02	1.17/2.23	1.35/2.31	1.44/2.80
		Multistage filters		
		d=2	d=3	d=4
MAG	0.34/1.48	0.24/7.78	0.16/10.2	0.12/12.5
MAG destIP	0.45/2.86	0.15/9.67	0.10/12.9	0.08/17.5
MAG ASpair	0.80/3.30	0.34/10.2	0.16/18.3	0.12/30.0
IND	0.95/2.27	0.35/14.0	0.17/15.9	0.17/21.4
COS	0.77/3.02	0.58/7.31	0.58/9.19	0.37/10.9

Table VI.2: The range of measured adjustment ratios

multistage filters than for sample and hold, especially as the number of stages goes up. A conservative way to choose the target usage would be the smallest ratio seen. Since the consequence of occasional memory overflows is not that severe (especially not for sample and hold using early removal, so most of the entries created towards the end of the measurement interval are not reported on anyway), we use the more aggressive value of 90% for traffic measurement devices using sample and hold and 85% for the ones using multistage filters.

Finding the right adjustment ratios

We used the same measurements as above to get minimum and maximum values for the adjustment ratio. We based it on the ratio of the average memory usage for consecutive thresholds each (approximately 40% apart). Table VI.2 contains our maximum and minimum values for the adjustment ratio over all thresholds and configurations. We also added the perfect knowledge algorithm (it decides which flows to add to the flow memory based on knowledge of their exact traffic) to be able to separate the effects of the peculiarities of the distributions of flow sizes from the behaviors introduced by our algorithms. We can see that sample and hold is much more robust than multistage filters (adjustment ratios closer to 1) and that it is very close (from this point of view) to the perfect knowledge algorithm. For certain settings (e.g., the MAG trace with flow ID

destination IP and an oversampling of 1) it is even more robust than the perfect knowledge algorithm. We can see that the robustness of sample and hold does not depend significantly on the oversampling factor. Based on these results we use a value of 1 for *adjustdown* and 3 for *adjustup* for traffic measurement devices using sample and hold. Multistage filters have huge maximum adjustment ratios, especially when the number of stages is large. This is because when filters are overwhelmed with traffic they quickly go from strong filtering to very little filtering. Based on the results we would use the following values for *adjustdown* and *adjustup*: 0.24 and 10 for 2 stage filters; 0.16 and 16 for 3 stage filters and 0.12 and 21 for 4 stage filters. However, after a number of sample runs it turns out that these adjustment ratios are too conservative, so we use an *adjustdown* of 0.5 and an *adjustup* of 3 instead.

VI.A.3 Adaptive bitmap

An adaptive bitmap achieves both the accuracy of a well-tuned virtual bitmap and the robustness of multiresolution bitmap by combining them. It relies on a simple observation: measurements show that the number of active flows does not change dramatically from one measurement interval to the other (so it is not suitable for tracking, say, attacks where sudden changes are expected). We use the small multiresolution bitmap to detect changes in the order of magnitude of the count, and the virtual bitmap for precise counting within the currently expected range. The number of flows we expect is the number of flows measured in the previous measurement interval. Assuming "quasi-stationarity", the algorithm is accurate most of the time because it uses the large, well-tuned virtual bitmap for estimating the number of flows. At startup and in the unlikely case of dramatic changes in the number of active flows, the multiresolution bitmap provides a less accurate estimate.

Updating these two bitmaps separately would require *two* memory updates per packet, but we can avoid the need for multiple updates by combining the two bitmaps into one. Specifically, we use a multiresolution bitmap in which r adjacent components are replaced by a single large component consisting of a virtual bitmap (where r is a configuration parameter). The location of the virtual bitmap within the multiresolution bitmap (i.e. which components it replaces) is determined by the current estimate of

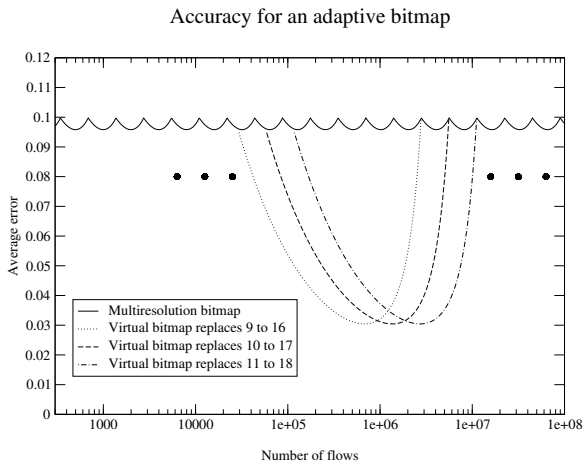


Figure VI.2: The large virtual bitmap replaces 6 of the components of the multiresolution bitmap. The size of the normal components is $b = 64$ bits and the size of the large virtual bitmap is $v = 1627$ bits. The adaptive bitmap guarantees an average error of at most 10% over the whole range, but if the number of flows falls into the "sweet spot" the average error can be as low as 3.1%

the count. If the current number of flows is small, we replace coarse components with the virtual bitmap. If the number of flows is large, we replace fine components with the virtual bitmap. The update of the bitmap happens exactly as in the case of the multiresolution bitmap, except that the logic is changed slightly when the hash value maps to the virtual bitmap component. The algorithm for computing the estimate for the number of active flows given the bits set in the adaptive bitmap is very similar to multiresolution bitmap. The main difference is that we use different threshold for selecting the big component as base. For brevity, we omit the algorithm.

Accuracy of adaptive bitmap

The error of the estimates of the adaptive bitmap depends strongly on the number of flows: the errors are much larger if the number of flows is unexpectedly large or small. The exact formulas, omitted for brevity are not very different from the ones from Section IV.D.2. We give an example instead. Figure VI.2 gives the average error as predicted by our formulae for the adaptive bitmap we use in for measurements

(Section VI.A.3). We first represent the average error of the original multiresolution bitmap and then the average error we obtain by replacing various groups of 8 consecutive components with the virtual bitmap. It is apparent from this figure that by changing which components are replaced by the virtual bitmap we can change the range for which the adaptive bitmap is accurate.

Configuring the adaptive bitmap

In this section we derive the minimum sizes for the large virtual bitmap used by the adaptive bitmap instead of the selected components. Since we already showed that a choice of $k = 2$ is optimal, we only consider this case for the adaptive bitmap. The reason we have a minimum for the number of bits in the large component is that we have to ensure that the error of the adaptive bitmap isn't worse than that of the multiresolution bitmap anywhere in the range of the number of active flows. The virtual bitmap does not influence our estimate when the number of flows is above the range it is responsible for because in this case the base component is after the virtual bitmap. If the base component is below the one just preceding the virtual bitmap then we are OK. The reason is that the bitmap is more accurate than just or-ing together the bits of all the components it replaces to the resolution of the coarsest of them, and this is at least as good as what our approximation formula uses. It will become apparent from later results that while the virtual bitmap covers the same interval of the hash space the or-ed together components would cover, it has more components than the coarsest component would if it were extended to cover that interval. All that's left to do is to choose it in a way that makes sure that our results are good when the virtual bitmap is the base component. Since we know that its accuracy will be worst at the end of the range we only need to compute its error there. When we look at the low end, we compare it against all the components added together; when we look at the high end, we compare it against just the finest component (because the multiresolution bitmap would ignore the others).

In Table VI.3 we report the costs and benefits of the adaptive bitmap. The first column lists the number r of normal components we replace with the large one. The next column lists the number of bits the large component needs to have (compared to

r	v/b	improvement
2	2.3626	1.1725
3	4.4861	1.4488
4	8.0603	1.8468
5	14.3252	2.4029
6	25.5510	3.1709
7	45.9411	4.2265
8	83.3330	5.6754
9	152.4217	7.6641
10	280.8654	10.3959
11	520.9068	14.1524
12	971.5300	19.3240

Table VI.3: As we increase the number of components r replaced by the virtual bitmap, the size of the virtual bitmap v almost doubles for each new component replaced. The ratio between the average error of the large virtual bitmap and the multiresolution bitmap also increases exponentially, but at a slower rate than the size of the virtual bitmap.

Trace	Adaptive bitmap (min/avg/max)	Probabilistic counting (min/avg/max)
MAG+	-4.402/1.066/4.717%	-9.525/2.820/13.262%
COS	-1.879/0.748/1.950%	-6.946/2.759/7.621%
IND	-1.767/0.601/1.772%	2.400/10.214/17.724%

Table VI.4: Comparison of adaptive bitmap and probabilistic counting errors, each using 16Kbits of memory

the number of bits of a normal component) to ensure that the adaptive bitmap never has a worse average error than the original multiresolution bitmap. The third column lists the ratio between the average error of multiresolution bitmap and the "sweet spot" average error of the adaptive bitmap.

Adaptive bitmap measurements

The experiments from this section compare adaptive bitmap and probabilistic counting on all three traces used in Section IV.E. The results are presented in Table VI.4. All of the algorithms were configured to use 16 Kbits of memory. For each algorithm we report the largest errors in both directions and the average error based on 20 runs with

different hash functions.

The algorithms were configured to give the best possible average error and work up to 100,000,000 flows. For the adaptive bitmap we used as a base a multiresolution bitmap with an average error of 10% with $k = 2$, $b = 64$, $c = 19$ and $b_{last} = 169$. The virtual bitmap component is $15,063$ bits large and replaces 9 components of the multiresolution bitmap. For the adaptive bitmap we did not include in our computations the first measurement interval when the adaptive bitmap was not tuned to the traffic. For the probabilistic counting we used $nmap = 744$ bitmaps of $L = 22$ bits each. Adaptive bitmap is roughly 3 times more accurate than probabilistic counting. For the IND trace, which has a very small number of active flows, probabilistic counting has very bad error and is actually biased towards overestimating. This is the same as the problem we noticed in the previous section. The major message here is that an adaptive bitmap can achieve almost the same benefits of virtual bitmap (e.g., order of magnitude reduction in memory for same accuracy) when the number of flows does not vary dramatically, as seems common in many networking applications.

VI.B Intra-interval adaptation

We can also adapt the sampling rate for sample and hold (PSH) within the measurement interval. The previous section controlled memory usage by adjusting the sampling rate from one measurement interval to the next based on the memory usage. This approach is vulnerable to memory overflowing during individual measurement intervals when the traffic mix changes suddenly. When the memory overflows, no new entries are created for the rest of the measurement interval, and important sources of traffic can go unnoticed. We aim to build a more robust adaptation mechanism that achieves good results even for the intervals where the traffic gets suddenly worse. In Section VI.A we used 5 second measurement intervals, not 5 minute intervals like we will use in this section, and this makes the problem of having incomplete data for a couple of measurement intervals less grave. We achieve robustness for these longer intervals by adjusting sampling rate within measurement intervals.

We first describe our algorithm for adapting the sampling rate within a mea-

surement interval assuming that we have a single table, say keyed on the source IP address, and a single algorithm, say PSH, creating the entries. Later we discuss how it can be generalized to multiple parallel sample-and-hold algorithms and to multiple tables.

Assuming that our system has enough memory to handle every packet of "normal" traffic without exceeding its memory limit, we begin each interval with the sampling rate set to 1. This means every packet is allowed to cause the creation of a new entry in the src IP table if that packet's src IP does not already exist in the table. Our first goal is to make sure that if the traffic mix changes, we stay within our available memory by decreasing the sampling rate. Furthermore, we want to achieve this goal with minimal loss of accuracy (for example, we could trivially ensure our first goal by setting the sampling rate to 0, but then our table would stay empty and provide no measurement results). Therefore we want to reduce the sampling rate no further than is necessary to ensure that we will not exceed our available memory.

One way of approaching this problem is to divide the measurement interval into multiple smaller subintervals, and based on the observed behavior in earlier subintervals adjust the sampling rates. This is not a robust solution because a sudden spike of malicious traffic could use all of the available memory before the current subinterval ends. We could defend against this problem by making the subintervals very small, but we are still vulnerable in the last few subintervals unless we put aside big chunks of the table as a safety buffer. Additionally, very small subintervals would be too sensitive to random short bursts in the traffic and cause us to adapt the sampling rate erratically.

Instead, we address the adaptation of sampling rates by dividing the available memory into smaller budgets. When the number of allocated entries reaches the current budget, we look at the rate at which entries have been allocated and decide whether we need to adjust the sampling rate: we decrease the sampling rate if and only if we expect the memory to run out before the end of the measurement interval, based on the growth rate of the table since the last rate adjustment. Thus when the traffic mix suddenly changes and PSH starts allocating entries very quickly, the table will exhaust the budget quickly and the algorithm will promptly reduce the sampling rate. In choosing the sizes of the budgets we need to balance two competing considerations: if they are too small

INIT_PSH

$interval_end = now + interval_size$

$abs_fill_time = now + 1.1 * interval_size$

$sampling_rate = 1$

$remaining_entries = available_entries$

$budget = remaining_entries/4$

$half_budget = remaining_entries/8$

$used_entries = 0$

$budget_start_time = now$

Figure VI.3: Initialization of per-table PSH state at the beginning of each interval.

the algorithm can overreact to small random spikes in the traffic, but if they are too large the algorithm will react too slowly. Furthermore, near the end of the interval we want to react more promptly because we have less memory left and unfriendly traffic can consume it faster. Our algorithm solves this problem by using budgets that are one quarter of the remaining available memory[1]. So the first budget is one quarter of the available memory, the next is one quarter of the remaining memory (three sixteenths of the total memory), and so on. Also, to avoid using very small budgets towards the end of the measurement interval, we perform the adaptation so that memory should run out slightly after the end of the measurement interval: When we adapt the sampling rate we pretend that the remaining time is 10% longer than it actually is, so that we will still have some memory left by the end of the actual memory interval. The size of the last budget will be no smaller than one quarter of this remaining memory which is the amount of memory we expect to fill up during the 10% "extension" to the measurement interval. Our final algorithm for PSH with adapting sampling rate is initialized at the beginning of each interval as shown in Figure VI.3, and then the code in Figure VI.4 is applied to each packet that does not already have an entry in the table. The purpose of $half_budget$ and $half_times[\,]$ will be explained later.

[1]We also experimented with other fixed fractions such as one half or one eighth, but one quarter seemed to offer the best balance between responsiveness and stability.

```
DO_PSH(packet)
  if random(0,1) ≤ sampling_rate
    CreateEntry(packet.srcIP)
    used_entries++
    if used_entries < half_budget
      return
    else if used_entries = half_budget
      half_budget_time = now − budget_start_time
      return
    else if used_entries < budget
      return
    endif
    half_times[1] = half_budget_time − budget_start_time
    half_times[2] = now − half_budget_time
    remaining_entries −= used_entries
    est_fill_time = predict_fill_time(half_times)
    fill_time = abs_fill_time − now
    if est_fill_time < fill_time
      sampling_rate ∗= est_fill_time/fill_time
    endif
    budget = remaining_entries/4
    half_budget = remaining_entries/8
    used_entries = 0
    budget_start_time = now
    time_remaining = interval_end − now
  endif
```

Figure VI.4: Algorithm for packet sample and hold on the src IP table with dynamically adapted sampling rate, applied to each packet whose src IP does not already exist in the src IP table. The algorithm works similarly on the other tables.

```
PREDICT_FILL_TIME(half_times[ ])
   if half_times[2] > half_times[1]
      slowdown = half_times[2] − half_times[1]
   else
      slowdown = 0
   endif
   oneeighthfilltime = half_times[2]
   timeleft = 0
   for i=3 to 8
      oneeighthfilltime+ = slowdown
      timeleft+ = oneeighthfilltime
   endfor
   return timeleft
```

Figure VI.5: Estimating the time it takes to fill up the other six eighths of memory based on the times it took to fill up the first two eighths (i.e. the two halves of the budget).

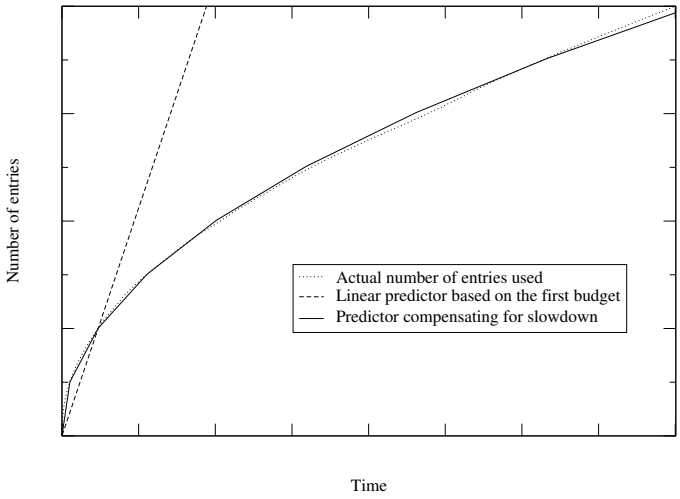


Figure VI.6: Using a linear prediction that assumes that the rate at which entries will be created is the same as the rate at which they were created during the current budget underestimates the time it takes to fill the memory. Accounting for the fact that it takes progressively longer to fill up the memory as we advance in time gives a much better prediction.

This adaptation algorithm needs to make a prediction of the rate at which table entries are going to fill up at the current sampling rate. This prediction is implemented by the "$predict_fill_time()$" function in Figure VI.5. While the prediction need not (and cannot) be exact, the adaptation is more efficient if the prediction is close to the actual behavior: predicting that the memory will run out a lot sooner than it actually would will prompt the adaptation algorithm to decrease the sampling rate unnecessarily, thus reducing the accuracy of the results, whereas predicting that the memory will run out much later than it actually would can consume the memory prematurely, forcing the algorithm to drastically reduce the sampling rate later on. Since we never increase the sampling rate within a measurement interval, we want to be especially careful that we do not severely underestimate the time it will take for the memory to fill up.

The simplest way of predicting when the memory will run out is to assume that the rate at which entries were allocated during the current budget period will continue. Figure VI.6 shows the number of entries created (without sampling) during

a typical measurement interval. The figure clearly shows that linear prediction is very far from reality, because the rate at which entries are created slows down as the time progresses because there are fewer and fewer new source IP addresses in the traffic mix. However, with higher sampling rates, the memory usage curve is much closer to a straight line. We need a simple predictor that works for both cases. Our predictor achieves this by measuring the rate of the slowdown in the memory usage: we measure the time it takes to use up the first and second halves of the budget and store these times in the previously mysterious $half_times[1]$ and $half_times[2]$. The difference between the two is the *slowdown*. We predict that the time it takes the algorithm to consume each eighth of memory is *slowdown* longer than the time to use the previous eighth. Therefore the third eighth should take $half_times[2] + slowdown$, the fourth should take $half_times[2] + 2 * slowdown$, and so on, and the total time to use up all six of the remaining eighths should be $6 * half_times[2] + 21 * slowdown$. Figure VI.6 also plots this new prediction, which is much closer to reality. Note that our prediction algorithm enforces that the slowdown is nonnegative (so it is never a "speedup"). If the second half of the budget is used up more quickly than the first half (due to an attack, for example), we use a slowdown of zero and thus base the prediction linearly on the rate at which only the second half of the budget was used.

VI.C Acknowledgments

This chapter is based on [EV03] which is joint work with George Varghese, [KME03] which is joint work with Ken Keys and David Moore and [EVF03] which is joint work with George Varghese and Mike Fisk.

VI.D Chapter summary

Usually the various components of traffic measurement systems operate with fixed resources. Due to the large variability of Internet traffic, any fixed choice of the configuration parameters can result in widely differing resource usage. Resource usage is also correlated with the accuracy of the results. In this chapter we examined methods to adapt the configuration parameters such as sampling rates and thresholds to the

traffic mix in order to obtain accurate results through a better utilization of the available memory. We found that well-chosen adaptation methods can result in significant improvements in accuracy without affecting the robustness of the measurement.

Chapter VII

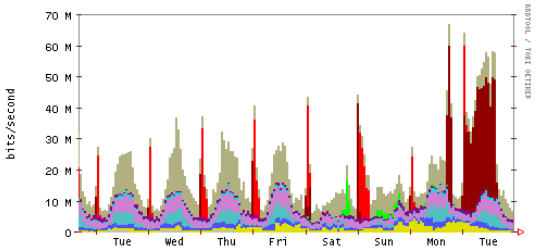
AutoFocus: a Flexible Traffic Analysis System

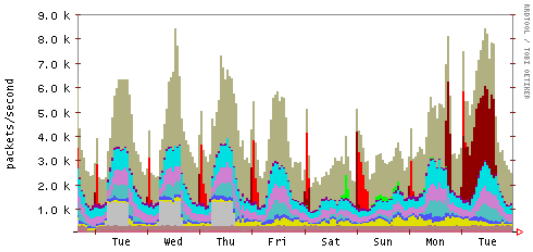
This chapter describes the design of our prototype system, AutoFocus, which uses reports based on traffic clusters described in Chapter V, and our experiences using it to discover the dominant and unusual modes of usage on several different production networks.

VII.A System description

The AutoFocus prototype is a traffic analysis system that can be used off-line or in semi real-time (it keeps up with traffic, but it is not real-time). It has three principal components:

- **Traffic Parser**. This component consumes raw network measurement data. Our current system uses (sampled) packet header traces as input, but it could easily be modified to accept other forms of data such as sampled NetFlow records.
- **Cluster Miner**. The cluster miner is the core of the tool and applies our multi-dimensional and unidimensional algorithms to compute compressed traffic reports, compressed delta reports and unexpectedness scores. The running time and memory usage of the algorithms that compute the traffic report depends significantly on the number of flows in the traffic mix. To reduce running time and memory usage






```
Total traffic is 354 GB. The threshold is 5%=17.7 GB.
Unidimensional reports
Protocol breakdown
 6(TCP)    98.653%  350 GB
Source IP breakdown
          137.131.0.0/16(Scripps)                              7.778%  27.6 GB
    192.128.0.0/10                                            66.450%  235 GB
                    192.172.226.64/26[Caida]                  61.371%  217 GB
                          192.172.226.89(magrathea.caida.org) 55.424%  196 GB
Destination IP breakdown
           132.249.0.0/17[SDSC]                               5.506%  19.5 GB
           137.131.0.0/17[Scripps]                            5.529%  19.6 GB
           137.131.128.0/17[Scripps]                          5.574%  19.8 GB
 192.0.0.0/8                                                 60.937%  216 GB
                          192.67.21.154(hpss07.sdsc.edu)     55.362%  196 GB
 198.0.0.0/8                                                  6.651%  23.6 GB
Source port breakdown
 lowport    22.768%  80.8 GB
 80(http)   16.784%  59.6 GB
 highport   76.532%  271 GB
 4339       55.362%  196 GB
Destination port breakdown
 lowport     6.121%  21.7 GB
 highport   93.178%  330 GB
 35904      55.362%  196 GB
Multidimensional report
 Source IP                          Destination IP                 Pr.  Src port   Dst port   Traffic   Label
 *                                  *                              TCP  lowport   highport   80.4 GB   108.2%
 *                                  *                              TCP  80(http)  highport   59.6 GB   108.8%
 *                                  *                              TCP  highport  lowport    21.1 GB   128.7%
 *                                  *                              TCP  highport  highport   248 GB     99.4%
 *                                  132.249.0.0/17[SDSC]           TCP  *         highport   19.0 GB   105.7%
 *                                  137.131.0.0/16(Scripps)        TCP  80(http)  highport   18.6 GB   306.0%
 *                                  137.131.0.0/16(Scripps)        TCP  highport  *          18.1 GB    60.9%
 *                                  137.131.0.0/17[Scripps]        TCP  *         *          19.5 GB   100.6%
 *                                  137.131.128.0/17[Scripps]      TCP  *         *          19.6 GB   100.6%
 *                                  192.0.0.0/8                    *    *         highport   214 GB    106.4%
 *                                  192.0.0.0/8                    TCP  *         *          214 GB    100.8%
 *                                  198.0.0.0/8                    TCP  *         highport   22.2 GB   102.2%
 137.131.0.0/16(Scripps)            *                              TCP  *         highport   20.5 GB    80.7%
 192.172.226.0/24(SDSC NAP)         *                              TCP  highport  highport   214 GB    139.9%
 192.172.226.0/25(Caida)            *                              TCP  *         highport   221 GB    110.5%
 192.172.226.64/26[Caida]           *                              TCP  *         *          217 GB    101.4%
 192.172.226.89(magrathea.caida.org) 192.67.21.154(hpss07.sdsc.edu) TCP  4339      35904      196 GB    596.7%
Delta report ------------ traffic changed from 213 GB to 354 GB, threshold is 5%=17.7 GB ------------
 *                                  137.131.0.0/17[Scripps]        TCP  highport  *          -20.6 GB
 134.79.0.0/18[SLAC.stanford]       137.131.0.0/16(Scripps)        TCP  highport  lowport    -19.6 GB
 192.172.226.3(ra.caida.org)        192.67.21.168(hpss45.sdsc.edu) TCP  highport  highport   -30.5 GB
 192.172.226.89(magrathea.caida.org) 192.67.21.154(hpss07.sdsc.edu) TCP  4339      35904      196 GB
```

Figure VII.1: The report for the 17th of December 2002 (one of the 31 daily reports for this trace) contains compressed unidimensional reports on all 5 fields and the compressed multidimensional cluster report using a threshold of 5% of the total traffic. In the unidimensional reports the percentages indicate the share of the total traffic the given cluster has. In the multidimensional report they indicate the unexpectedness score. Note how much smaller the delta report is than the full report.

without loosing much accuracy, AutoFocus uses traffic synopses (Section V.E).

- **Visual Display**. The visual display component is responsible for formatting the

report and constructing graphical displays to aid understanding. To improve user recognition of individual elements, we post-process the raw traffic report to attach salient names to individual addresses and ports. These names are generated from the WHOIS and DNS services, lists of well-known ports, as well as user-specified rules that contain information about the local network environment (e.g., that a particular host is a Web proxy cache or a file server). The display component also generates a series of time-domain graphs, using different colors to identify a set of key traffic categories. These categories can contain multiple clusters. Categories are ordered and each flow is counted against the first category it matches, traffic not falling into any particular category is lumped into an "Other" category. Ideally, the categories are also constructed to be representative of "interesting" aggregates (e.g., outbound SSL traffic from our Web servers). Currently, the user specifies these categories – typically based on examining the clusters contained in the textual report. Heuristics for automatically selecting these traffic categories remain an open problem, complicated by the requirement that categories be meaningful. We expect that some user involvement will always be beneficial.

Figure VII.1 depicts a report generated by AutoFocus using a 5% threshold on a trace recently collected from the SD-NAP exchange point. After identifying the size of the total traffic and the threshold parameters, the report provides the five unidimensional compressed reports – protocol, source address, destination address, source port and destination port. For example, the report indicates that 66% of traffic in this trace originates from 192.128.0.0/10, however most of this traffic can be attributed to the more specific prefix, 192.172.226.64/26 (owned by CAIDA). Note that prefixes between these two records, from /10 to /26, are compressed away because their traffic does not differ by more than 5% (17.7GB) from the more specific /26 prefix. Ultimately, an individual source IP address is responsible for the majority of this activity. The compressed multidimensional report starts with the least specific clusters (such as arbitrary TCP traffic from servers using low ports to clients). Note that the most specific cluster shown exactly identifies the *particular transfer* that consumed more than half of the total bandwidth. Moreover, its unexpectedness score of 596%, promptly brings it to the attention of the network administrator. In addition, this cluster is also identified in the delta section at

the end of the report. The output of AutoFocus also includes time-series plots of traffic as bytes and packets colored by the appropriate categories on two timescales: short (two days – not shown here) and long (eight days). The conspicuous spikes at each midnight are periodic backups.

The AutoFocus prototype has another feature that proved useful on a number of occasions: *drilling down* into individual categories. For each of the categories, we provide separate time series plots and reports that analyze the internal composition of the traffic mix within that particular category.

VII.B Experience with AutoFocus

VII.B.1 Comparison to unidimensional methods

We contrast our multidimensional method with the unidimensional analysis. Figure VII.2 presents a simplified version of the time domain plot generated by AutoFocus for Friday the 20th and Saturday the 21st of December 2002, while Figures VII.3 to VII.7 present the five unidimensional plots. Besides compactness, our multidimensional view has the advantage of making it easier to see very specific facts about the network traffic. For example the light colored "spot" between 7 AM and 2 PM on the first day is UDP traffic that goes to a specific port of a specific multicast address. While we could manually correlate the corresponding "bright spots" from the protocol, destination prefix and destination port plots, the AutoFocus plot automatically identifies the key usage directly. The massive spike causing the traffic surge from 12:01 AM until 3 AM the first day and the longer dark traffic cluster from 1 AM until 11 AM that day and 1 AM and to 6 AM the second day are two different types of backups. It would be difficult to disentangle them using only unidimensional plots. Since they use different source ports (SSH versus high ports) they show up separately in the source port report, but since they come from the same source network and go to the same destination network they show up together in those plots.

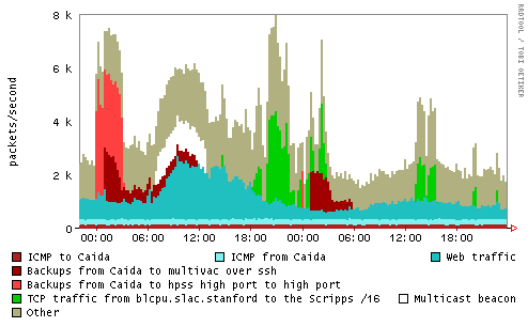


Figure VII.2: Multidimensional report for Friday the 20th and Saturday the 21st of December 2002 using traffic clusters

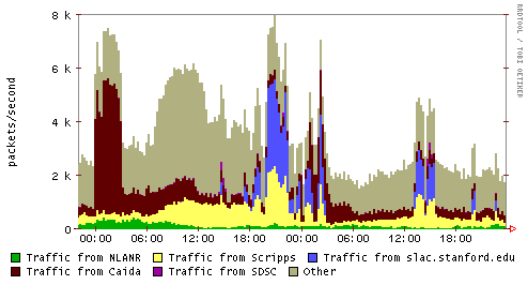


Figure VII.3: Unidimensional report for source network

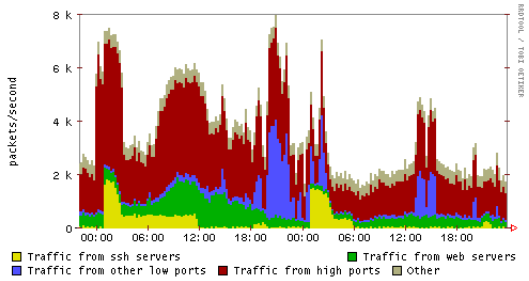


Figure VII.4: Unidimensional report for source port

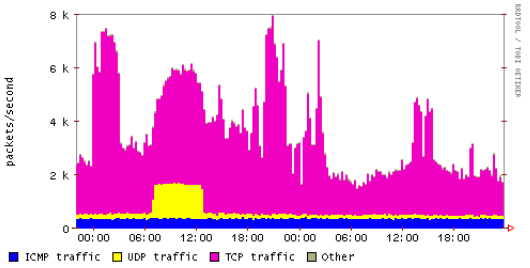


Figure VII.5: Unidimensional report for protocol

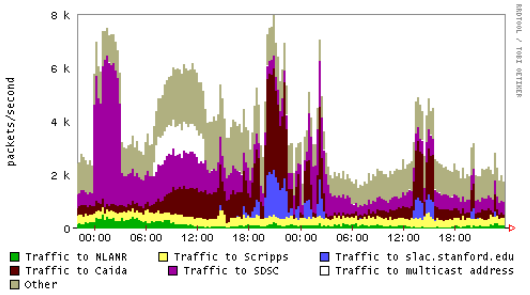


Figure VII.6: Unidimensional report for destination network

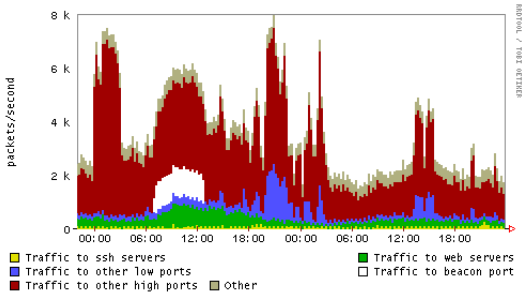


Figure VII.7: Unidimensional report for destination port

VII.B.2 Experience with analysis of traffic traces

This section presents highlights of our experience using AutoFocus to analyze traffic traces from three large production networks. While the usefulness of the insights gleaned using AutoFocus is hard to quantify, we hope that presenting some of them will give the reader a more accurate idea of the power of our system.

Small network exchange point

Our first trace was collected from SD-NAP, a small network exchange point in San Diego, California, that interconnects many research and educational institutions and also connects some of them to the rest of the Internet. The trace is 31 days long and it starts on the 7th of December 2002. In addition, we were able to consult with those familiar with this network to calibrate our conclusions and receive useful feedback about our results.

When analyzing the raw reports we looked for traffic clusters that were large, but did not completely dominate the traffic (e.g., the cluster containing all TCP traffic is not very useful). In particular, we found that port 80 (TCP) Web traffic was so large that it was best subdivided into four categories: Web traffic from a particular server at the Scripps Institute, Web traffic destined for clients within the Scripps /16 (Web client traffic), other traffic from port 80 and traffic to port 80. Of these, the first and second categories proved to be the most interesting. Traffic from the Web server showed a clear diurnal pattern with peaks before noon (sometimes a second peak after noon) and lows after midnight. It also showed a clear weekly pattern with lower traffic on the weekends and holidays. The second category (the Web clients) had similar trends, but the lows around midnight usually went down to zero and the difference between weekend and weekday peaks was much larger. In retrospect, this is to be expected since the client traffic requires the physical presence of people at Scripps, while the server traffic can be driven by requests from home machines or users outside the institution.

Another interesting traffic cluster contained Web proxy traffic originating from port 3128 (used by Squid web caches) of a particular server at NLANR. The amount of traffic had a daily and weekly cycle, but the lowest traffic was at noon and the highest at midnight. Using AutoFocus' drill-down feature we were able to examine the breakdown

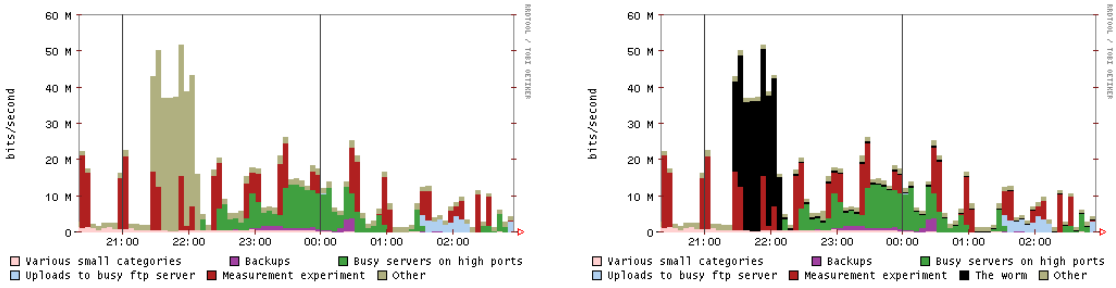


Figure VII.8: The Sapphire/SQL Slammer worm shows up in the time series plots as a big increase in the traffic of the "Other" category around 21:30. Once we highlight the worm by putting it into a separate category, it is evident that while its traffic is significantly reduced at 22:10 when the infected internal hosts were neutralized, worm traffic persists at a lower level because of outside hosts spreading it into this network.

of this traffic, which identified large clusters containing transfers to second-level caches from Taiwan, Indonesia, Spain and Hong Kong. Evidently the traffic was driven by the daily cycle of the clients in these other time zones.

AutoFocus places the remaining non-categorized traffic into an amalgamated "Other" category. On the last day of the trace, we saw a huge increase at 5:30 PM in the "Other" traffic that saturated the link followed by a sudden decrease at 11 PM. Again using the drill-down feature, we observed that this change was mostly attributable to traffic between two nearby universities: from UCSD to UCLA. However, traffic between these two did not appear in any other cluster over the previous 30 days. Upon investigation we determined that the traffic bulge was due to a temporary network outage that forced traffic normally using the CalREN network to transit SD-NAP instead.

Large research institution

Our second trace was taken from the edge of a network that connects a large research institution (roughly 9,000 hosts) to the Internet. The trace is 39 days long and it starts on the 12th of December. In this case, we had access to similar although less-detailed expertise concerning the operation of the network.

Many findings for the second trace were similar to those we have described earlier, but there were some differences. We observed a series of regularly scheduled

backup transfers: one from a range of machines destined to TCP port 7500 that had regular daily spikes at 11PM and 5AM followed by periods of quiescence. Another example was a regular 40GB transfer that started at 8PM on each Wednesday (usually lasting until 10AM the following morning). This activity proved to be a full backup of a large RAID array.

We observed a single large cluster containing a series of regular TCP transfers from a single host to port 5002 on three other hosts distributed around the Internet. This turned out to be a regularly scheduled network measurement experiment that was part of a distributed research activity.

The most interesting result for this second trace came by looking at the breakout of the Sapphire worm [MPSS+02]. This worm exploits a vulnerability of the Microsoft SQL server running on UDP port 1434 and spreads extremely aggressively using single 404 byte packets to infect random destinations. We computed the traffic reports for three hour measurement intervals. The time at which the worm started is apparent in the time series plot (see Figure VII.8). It shows up as a huge increase in the "Other" traffic category. Drilling down into the report describing that category, the worm is conspicuous: 90% of the traffic was to UDP port 1434. A quick comparison between the packet and byte reports also gives us the average packet size. Furthermore, the report readily revealed the 6 internal IP addresses that generated 80% of the traffic: these were local hosts infected by the worm aggressively trying to infect the outside world. These compromised servers were promptly neutralized by the network administrators. In the next three hour interval, while the traffic in the "Other" category decreased to levels similar to those before the worm, still a substantial fraction of it (23%) was worm traffic. The report revealed that this was traffic originating on the outside: it consisted of incoming probes trying to infect internal hosts. We also performed an analysis of the trace with the worm traffic separated into its own category (the second plot from Figure VII.8). We were able to see fine details. For example, some of the infected hosts did not spread their traffic uniformly over the whole address space, but focused on single /8s. This is consistent with the observation [MPSS+02] that for some values of the random seed, the algorithm used by the worm to select target addresses chooses them from a limited set. This example shows once again the strength of our multidimensional

approach: AutoFocus is able to promptly bring to the network manager's attention and describe in great detail such unexpected and unpredictable event as a worm epidemic.

While none of the traces we worked with contained massive denial of service attacks, we believe that AutoFocus would bring them to the attention of the network operator the same way it showed the worm. The victim of the attack (whether it is an individual IP address or a prefix) will show up in the report with a very large number of packets (or bytes depending on the type of attack). Furthermore, the attack will reveal the protocol used by the attack and possibly the port number if it is kept constant. If the source address is faked at random from the whole IP address space, the report will not associate any particular source address with the attack traffic hitting the victim. However, if for some reason (e.g., egress filtering at the site the attack originates from), the source addresses are restricted to a certain prefix (or a small number of prefixes), the report will identify these, thus facilitating prompt and specific response.

We presented the output of AutoFocus to network managers of the first two networks we had traces from. Their reactions were very positive. It was easy for them to understand the output. They appreciated the intuitiveness of the time series plots, the large amount of information they convey and the ease with which the traffic reports and the drill-down feature provided them more detailed information when they needed it. The managers of both networks expressed interest in deploying our tool.

Backbone

A third trace we looked at was captured in August 2001 from an OC-48 backbone link and is 8 hours long. We looked at traffic reports for one hour measurement intervals. The reports reveal that around two thirds of the bytes on the link come from TCP port 80 and around one third come from high ports. The report also revealed that around one third of the traffic was from high ports to high ports. This is consistent with the behavior of peer-to-peer traffic. Through the unexpectedness scores, the report revealed some further facts that seemed surprising at first. There were specific source and destination prefixes where the Web traffic represented almost all of the traffic. One explanation for the prefixes that send almost only Web traffic is the clustering of Web servers in Web hosting centers or server farms. The prefixes that receive almost exclu-

sively Web traffic could be organizations with many Web clients whose internal policy prohibits the use of peer to peer applications.

VII.C Acknowledgments

This chapter is based on [ESV03] which is joint work with George Varghese and Stefan Savage. We would like to thank David Moore, Vern Paxson and Mike Hunter for their help with the evaluation of the AutoFocus prototype.

VII.D Chapter summary

We implemented the traffic cluster algorithms from Chapter V in the AutoFocus traffic analysis system. We have developed a Web-based user interface to allow managers to explore clusters across multiple time-scales and to drill down to explore the contents of any clusters of interest. Our preliminary experiences with this tool have been extremely positive and we have been able to identify unusual traffic patterns that would have been considerably harder to identify using conventional tools. Moreover, we have received positive feedback from network mangers, who have quickly appreciated the benefits of our approach.

Finally, note that AutoFocus, as described in this chapter, automatically extracts patterns of resource consumption in a *single interval* of time based on the traffic log of a *single link*. The natural generalization would to extend AutoFocus to automatically extract patterns of resource consumption *across time* and *across space*. While AutoFocus currently provides a *visual* display across a limited number of periods, it would be useful to do *automatic* time-series analyses across large time periods using compressed clusters as a new and parsimonious basis for such analysis. Similarly, extending AutoFocus to detect resource consumption *across space* would allow managers to detect geographical patterns within the network. We leave these generalizations for future work.

Chapter VIII

Conclusions

More than a hundred pages ago we set out to make the world a better place by improving Internet traffic measurement. It is time we look back and candidly assess how well we succeeded. Table VIII.1 extends our table of the existing traffic measurement solutions, Table II.1, with some important solutions proposed in this dissertation. While the table suggests that we achieved some progress, the sections to follow will give a more detailed account.

VIII.A Summary of dissertation

The paradigm at the foundation of this thesis is identifying building blocks common to many traffic measurement applications and finding good algorithmic solutions for them. Experience seems to confirm that this is a fundamentally sound approach and applying it makes system design and the evaluation of the final system significantly easier. AutoFocus (Chapter VII), EarlyBird [SEVS03] and the robust traffic summaries [KME03] are traffic measurement systems with widely differing goals, but they all use the building blocks we identified. Figure VIII.1 shows how good building blocks can keep resource limitations from turning into bottlenecks for the traffic measurement process.

Our first algorithmic building block, presented in Chapter III, is identifying and measuring large flows. We have two solutions, both suited for hardware implementations. These algorithms can be applied at all stages of the traffic measurement system, but are

Solution	Flexible analyses	Auto-mation	Info. content	Output size	Memory used	Reada-bility
SNMP counters		***	*	****	****	*
MRTG plots		***	*	****	***	****
Tomography TM		***	**	***	***	**
tcpdump(traces)	****	***	*****		****	
NetFlow	***	***	****	*		
Sampled NetFlow	***	**	**	**	*	
Aggr. NetFlow	*	***	**	**		*
Flow sampling	***	**	***	***		
SFlow	***	***	**	*	****	
top K reports		***	**	****	*	****
FlowScan	*	**	**	****	**	****
ACLs	***		***	****	***	**
sampl.&h.+adapt. Chapters III,VI	*	***	***	****	***	***
m.s.filter+adapt. Chapters III,VI	*	**	***	****	***	***
flow count bmps. Chapter IV		***	*	****	****	****
cluster reports Chapter V	**	***	**	****	**	****

Table VIII.1: Traffic measurement solutions have strengths (indicated by many stars) and weaknesses. The first column evaluates the diversity of the analyses supported by the output of the given system, the second looks at the human effort required to configure a given solution and the third at the information content of the results (including accuracy).

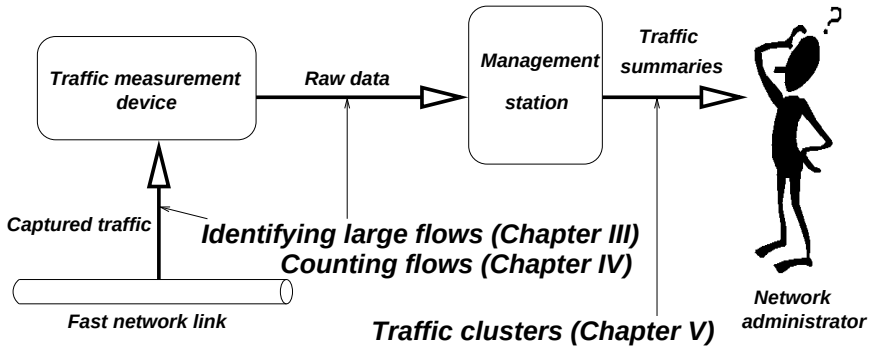


Figure VIII.1: The algorithmic solutions to the building blocks we identified help relieve potential bottlenecks in the traffic measurement process

particularly valuable at the early ones (Figure VIII.1). Our first algorithm, sample and hold, uses random sampling to identify flows to track and keeps per-flow counters for the flows it tracks in a small flow memory. Multistage filters uses multiple stages of counters to which packets are hashed independently and only allocates entries in the flow memory for packets that hash to counters above a threshold, thus filtering out most small flows. To the best of our knowledge, much of our presentation of these algorithms is novel in the whole computer science literature: our analysis of parallel multistage filters, serial multistage filters, shielding, preserving entries, early removal and filters with leaky buckets. We'd like to specifically point out conservative update, which improves filtering significantly by not incrementing any of the counters beyond the size of the smallest of them, which is a lower bound on the size of the flow the current packet belongs to. Furthermore, our experiments and the heuristic rules for dimensioning measurement devices based on our algorithms will be useful for future systems using them. Though multistage filters have better performance than sample and hold, due to the extreme simplicity of sample and hold we expect it to be preferred in future systems that need to identify large flows.

Our second algorithmic building block, presented in Chapter IV, is counting active flows and the related problem of identifying and measuring the sources or destinations with many flows. We have a family of bitmap algorithms that address this problem. They all hash packets to a bitmap based on a hash of their flow identifier and compute an estimate for the number of flows based on the number and position of the bits set. The details of the mapping of flow identifiers to bits differ and so do the functions for computing estimates for the number of flows based on the bits set. Since the hashing and mapping functions are simple and the per-packet processing very limited, all our bitmap algorithms are ideally suited for high speed hardware implementations. While basic algorithms such as direct bitmap and virtual bitmap have been known before, a unified presentation that includes them in the larger family of our bitmap counting algorithms facilitates future work on related algorithms customized to the specific needs and idiosyncrasies of the systems they are part of. Adaptive bitmap, triggered bitmap and increment/decrement algorithms are a good example of the flexibility of this family of algorithms. We consider multiresolution bitmaps to be an important advance as they

can provide consistently accurate counts for a very wide range for the number of flows and thus might be the choice for many traffic measurement systems. Flow sample and hold extends the idea of sample and hold to identify sources or destinations that have many flows, not many packets or bytes. Due to its effectiveness and simplicity of implementation and configuration, flow sample and hold might become a popular method for identifying sources or senders with many flows.

The third algorithmic building block, presented in Chapter V is a new one: explicitly describing traffic mixes. It generalizes the idea of identifying and measuring large flows to aggregates called traffic clusters, defined through an arbitrary subset of the important fields in the packet header, at an arbitrary granularity. The hardest benefit to quantify of this building block is how easy the explicit descriptions are to read by humans, which makes it especially well-suited for late stages of traffic measurement systems (Figure VIII.1). Traffic reports accurately describe all traffic clusters above a given threshold and achieve conciseness through our novel compression rule. Our multidimensional compressed reports are similar in spirit, yet quite distinct from, the association rules used in data mining [AIS93] and the data cube [GCBL+97] used in databases. The unexpectedness labels we include in the reports proved very helpful for identifying the large non-obvious aggregates in the traffic. The compressed delta reports concisely describe the difference between two traffic mixes and are thus perfectly suited for the important task of identifying changes in the traffic mix.

In Chapter VI we explore a technique that our experiments show to be very effective at increasing the accuracy of the results of various components of the system, without exceeding strict resource limitations: dynamically adapting parameters to the traffic mix. The large variability of traffic mixes makes adaptation especially useful. It eliminates the need for choosing fixed values for parameters such as many sampling rates, thus reducing the configuration overhead of traffic measurement systems. It also gives them a robustness in the face of adverse traffic mixes that many current solutions sorely lack.

AutoFocus, presented in Chapter VII, is an automatic traffic analysis system based on multidimensional traffic cluster reports. It prompted many insights into the traffic mixes we analyzed with it. AutoFocus combines the textual traffic reports with

easy-to-read time series plot of the traffic, with important categories of traffic (based on traffic clusters) highlighted. The user interface implements a lightweight filtering operation that together with traffic reports precomputed at many thresholds allows the network administrator to easily explore finer aggregates within the traffic mix without being overwhelmed by unnecessary detail. Filtering, together with drill downs into traffic categories and reports at multiple timescales, provides a rich environment network administrators can navigate to find out about their traffic. The initial feedback from network operators who used early versions of the system has been very positive and gives us reason to believe that more mature versions of the system will enjoy popularity in the near future.

Table VIII.2 summarizes the contributions of this dissertation. The columns denote the three ways in which network measurement research can be valuable: by extending the field of computer science (or, more generally speaking, humankind's knowledge) with new intellectual artifacts or new bits of knowledge about the network; by supporting future developments in the field; by improving in some concrete way the operation of computer networks. These are not metrics we can accurately measure. Since the amount of technical literature published in the field of computer science is much beyond the capacity of any single human to absorb, novelty can never be exactly measured, but a search of the literature coupled with discussions with knowledgeable researchers from the relevant fields can lead to a fairly clear conclusions. The situation is much murkier with the two types of impact. Since impact is measured through future work and this work has been only recently published, all we can provide at this stage is an informed, honest guess. It would be more accurate to call the columns estimated or expected impact.

VIII.B Future directions

Having summarized the contributions of this dissertation and having highlighted their wide applicability, it is exciting to discuss future directions this work can lead to. It is our hope that having identified building blocks and provided efficient algorithmic solutions, future systems that rely in some way on traffic measurement will be

Paradigms	Chapters	Novelty	Research impact	Network impact
Common algorithmic building blocks	I,III,IV,V		+	+
Direction of aggregation based on traffic	V,VII	++	++	++
Adapting algo. parameters to traffic mix	VI		+	+

Algorithms	Section	Novelty	Research impact	Network impact
Sample and hold	III.D.1		++	+
Parallel multistage filters	III.D.2		+	
Serial multistage filters	III.D.2	+		
Leaky bucket multistage filters	AppC	+		
Preserving entries, shielding, early removal	III.D.3	+		+
Conservative update	III.D.3	++	+	
Parallel multistage filter analysis	III.E.2	+	+	
Experience and dimensioning rules	III.G	+	+	
Direct and virtual bitmaps	IV.C		+	+
Multiresolution bitmap	IV.C.3	++	+	+
Adaptive bitmap, triggered bitmap	IV.C	+		
Flow sample and hold	IV.C.5	+	+	+
Increment/decrement algorithms	IV.C.6	+	+	+
Unidimensional compressed reports	V.C.1	+		+
Multidimensional compressed reports	V.C.2	++	++	++
Unexpectedness labels	V.C.2	+	+	+
Compressed delta reports	V.C.2	+	+	+
Traffic report filter	VII.A		+	+

Table VIII.2: Summary of contributions of this dissertation

easier to build and their performance will be better understood. For example our building blocks could be used to build a system that identified worms based on their traffic patterns, without relying on worm signatures provided by humans [SEVS03]. Worms infect vulnerable computers and try to spread aggressively. Thus in the process, it is inevitable that they send frequently the code snippets exploiting the vulnerability. Our algorithms for identifying large flows can be used to find these snippets repeating often (by defining flows based on packet contents, not based on packet header fields), and we can use their results to build signatures for new worms. But there are many legitimate strings that repeat often in the traffic, so we need further criteria for distinguishing worms. One such criterion is that during a worm infection, many infected computers are trying to spread the worm. Our bitmap algorithms offer a good solution for counting the number of sources associated with each suspected signature.

There are bolder steps to be taken. New building blocks will probably be identified, further extending the library assisting system developers. Our building blocks focus on counting bytes, packets and flows. We believe there are many interesting new ways to measure the traffic. In this dissertation we focus on measuring the traffic of a single link. It would be interesting to see how our methods could be extended to a whole network. We divide the time in measurement intervals and ignore any timing of the packets within them. We are convinced there is useful information to be extracted from the fine timing of the packets within the interval.

Traffic cluster reports characterize the traffic mix based on simple hierarchies of the five important header fields. Using more elaborate hierarchies, say based on routing tables, AS numbers and BGP community strings can provide more useful results. There are more radical ways of moving forward. We can apply multidimensional cluster analysis to application level data: (web) server logs or protocol level data reconstructed from live traffic by intrusion detection systems such as Bro [P99]. Synopses and traffic cluster reports describe the composition of the traffic associated with each large source or destination (network). One could use advanced data mining techniques to compare these traffic fingerprints and classify them into meaningful categories that distinguish say between server farms, dial-up ISPs, broadband providers and sites running scientific computing applications with massive data transfers. An interesting related problem is

to automatically divide the traffic mix into distinct categories, an activity that is now performed manually.

The compressed multidimensional reports based on traffic clusters might be the beginning of a line of research with implications beyond networking into data mining and knowledge discovery. These reports are a compact and explicit synopsis data structure that can describe the data set in great detail. The ease with which we were able to add drill-down through filters suggests that compressed multidimensional reports or other related structures are also well suited for user driven exploration of the data. More efficient randomized algorithms, possibly operating in streaming fashion, do probably exist for computing these compressed reports or other, equally strong, related forms. These structures could probably be extended with many useful operations, especially if the error in their approximation of the original data set stays quantifiable.

To give more perspective to these predictions, we can abstractly contrast multidimensional compressed reports with two well established areas of research: association rules and On-Line Analytical Processing OLAP (inspired by the data cube [GCBL+97]). All three approaches incorporate natural hierarchies. Since OLAP seeks to answer SQL queries exactly, unlike association rules and compressed traffic reports which focus only on large sets, it maintains an exact copy of the data set (and actually extend it with precomputed data). Paying this cost is not feasible in all settings. Since the structure of OLAP data is dictated by the structure of the space the data set is collected from, not the actual data set, exploration in OLAP might be more tedious and less directed. Unlike association rules, both our approach and OLAP have a small number of fields associated with all data items, which allows solutions that could not work with a large number of fields. Thus, while related to these two areas of research, our approach is not only different, but it also has specific advantages, and this strengthens our belief that it may be the origin of a vigorous new line of research.

Appendix A

Cisco NetFlow

NetFlow [CN] is a feature of Cisco routers that implements per flow traffic measurement. It is one of the primary tools used to collect traffic data by large transit ISPs today [FGLR+00]. NetFlow is intended (by Cisco) to serve as a basis for usage based billing. We briefly discuss here some details of Cisco NetFlow. We also present an analytical evaluation of the accuracy of sampled NetFlow and its memory requirements. At the end of this appendix we propose an alternative implementation solution that could increase by an order of magnitude the link speeds NetFlow can handle without resorting to sampling. This implementation procedure can also be used in conjunction with our algorithms.

A.1 Basic NetFlow

NetFlow defines flows as unidirectional streams of packets between two particular endpoints. A flow is identified by the following fields: source IP address, destination IP address, the protocol field in the IP header, source port, destination port, the TOS byte and the interface of the router that received the packet. In the DRAM of the router interface card there is a flow cache that stores per flow information (we call it flow memory in Chapter III). The entry for a flow holds, besides the flow identifier, various types of information about the flow: timestamp of when the flow started and ended, packet count, byte count, TCP flags, source network, source AS (Autonomous System), destination network, destination AS, output interface, next hop router. Various heuristics

(e.g. flows that have been inactive for a particular period of time, the RST and FIN TCP flags) are used to determine when a flow ends.

The NetFlow data captured by the router is exported via UDP packets to computers that process it further. The raw NetFlow data can be processed in a variety of ways and can give all kinds of information about the traffic. There are two major problems with the basic NetFlow: for interfaces faster than OC3 updating the flow cache slows down the operation of the interface and the amount of data generated by NetFlow can be so large that it overwhelms the collection server or its network connection ([FGLR+00] reports loss rates of up to 90%). Cisco's solution to the first problem is sampling packets and to the second aggregating the measurement data on the router.

A.2 NetFlow Aggregation

Many applications are not interested in the raw NetFlow data, but in an aggregated form of it. For example, when deriving traffic demands one is interested in traffic between networks (more exactly IP prefixes), not individual endpoints: all NetFlow records of individual flows whose two endpoints are in the same two networks are aggregated together. One can also imagine arrangements between ISPs with payment based on traffic that would require a similar type of aggregation.

Cisco's solution to the problem of NetFlow generating too much data was introduced in IOS 12.0(3)T . The aggregation of raw data is performed at the router. One or more extra caches called aggregation caches are maintained at the router. Only the aggregate data is exported thereby reducing substantially the amount of traffic generated. Five aggregation schemes are currently supported: based on source and destination AS, based on destination prefix, based on source prefix, based on source and destination prefix and based on source and destination ports.

A.3 Sampled NetFlow

Cisco introduced a feature called sampled NetFlow [CSN] with high end routers. The performance penalty of updating the flow cache from DRAM is avoided by sampling the traffic. For a configurable value of a parameter x, one of every x packets

is sampled. The flow cache is updated only for the sampled packets. Even though the update operation is not performed any faster, since it is performed less often it does not affect the performance of the router. Cisco recommends that sampling is turned on for interfaces above OC-3. The advantage of this solution is that it is very simple and requires no significant changes to the hardware of the line card.

A.4 The accuracy of sampled NetFlow

The actual sampled NetFlow works by counting every x-th packet irrespective of packet sizes. To simplify the analysis we will assume that all packets have the same size y and are sampled with probability $p = 1/x$.

Let c be the number of packets counted for a given flow and s the actual size of the flow (in packets). The probability distribution of c is binomial. The probability that a flow of size s is missed is the same as the probability that no packets get sampled which is $(1-p)^s$. By the linearity of expectation we obtain that $E[c] = sp$. Therefore the best estimate for s is c/p. Since the probability distribution for c is binomial, its standard deviation will be $SD[c] = \sqrt{sp(1-p)}$. The standard deviation of our estimate of s will be $1/p\sqrt{sp(1-p)}$.

To compare the accuracy of sampled NetFlow with our algorithms we compute the standard deviation of the estimate of the size of the flow that is at the threshold $T = s * y$ (in bytes). By substituting in the formula above, this is $y/p\sqrt{p(1-p)T/y} = \sqrt{y(1-p)T/p}$. Based on this number we can also compute the relative error of a flow of size T which is $\sqrt{y(1-p)/Tp}$. We can substitute actual numbers into this formula. Since sampling is recommended above OC-3 (155.52 Mbits/s=19,440,000 bytes/s), if the line speed is x times OC-3, then the sampling probability is at most $p = 1/x$. Smaller sampling probabilities can be used to reduce the memory requirements at the cost of accuracy. Let the measurement interval be i seconds. Assuming a threshold of $T = zC = x\ i\ z\ 19,440,000$ and a packet size of 1500 bytes (which is common for large flows), the relative error of the estimate of a flow at the threshold is $\sqrt{1500(1-1/x)x/T} \approx \sqrt{1,500/(19,440,000\ i\ z)} = 0.0087841/\sqrt{zi}$.

A.5 The memory requirements of sampled NetFlow

To be able to compare NetFlow to our algorithms for identifying large flows, for the purpose of this analysis we change somewhat the way NetFlow operates: we assume that it reports the traffic data for each flow after each measurement interval, like our algorithms do. The number of entries used by NetFlow is bound by both the maximum number of packets sampled during a measurement interval and the number of active flows n. Assuming the link is fully utilized with minimum size packets of 40 bytes, the number of packets sampled in i seconds is exactly $ipC/40$. As we saw in Section A.4, the maximum sampling that doesn't slow down the packet forwarding is $p = 19,440,00/C$. If we use this sampling rate, the maximum number of updates per measurement interval is $i\ 19,440,000/C\ C/40 = 486,000\ i$.

A.6 Keeping a queue of packet headers

The improvement presented in this section significantly increases the amount of time NetFlow can spend with each packet. It involves addition of a simple SRAM buffer.

In [LS98] Lakshman and Stiliadis argue that packet forwarding and classification decisions have to be made at line speed even for the smallest of packets. We argue that this does not extend to traffic measurement. We can keep the packet headers and other relevant information in a small queue and process that information (for traffic measurement purposes) at somewhat lower speeds after the packet was sent on the wire. This does not cause any delay for the actual packet. We are basically decoupling the forwarding of packets from the traffic measurement device. We argue that the benefits far outweigh the costs of this improvement.

Practically all of the packets from the traces we used are at least 40 bytes large. However the average size is around 550 bytes. If we were to dimension the traffic measurement device to handle at line speeds packets of 240 bytes instead of 40 bytes, this would give us 6 times as much time to process each packet. Since the average time the traffic measurement device has to process a packet is more than twice what it needs, the SRAM buffer holding the queue of packet headers need not be very large to make

it very unlikely that it ever overflows, but is must sill be large enough to absorb bursts of minimum sized packets. This is very similar to how packet headers are buffered on cards used for traffic capture until the driver can handle them.

Appendix B

Analysis details for algorithms for identifying large flows

B.1 Details of the analysis for multistage filters

This appendix presents the full derivation for Theorem 3. We use the same notation as in Section III.E.2. We first derive the necessary lemmas. We also give two high probability bounds on the number of flows passing the filter: a loose bound that has a closed form and a tighter one we specify as and algorithm.

Lemma 10 *The probability of a flow of size $s \geq 0$ passing one stage of the filter is bound by $p_s \leq \frac{1}{k}\frac{T}{T-s}$. If $s < T\frac{k-1}{k}$, this bound is below 1.*

Proof Let's assume that the flow is the last one to arrive into the bucket. This does not increase its chance to pass the stage, on the contrary: in reality it might have happened that all packets belonging to the flow arrived before the bucket reached the threshold and the flow was not detected even if the bucket went above the threshold in the end. Therefore the probability of the flow passing the stage is not larger than the probability that the bucket it hashed to reaches T. The bucket of the flow can reach T only if the other flows hashing into the bucket add up to at least $T-s$. The total amount of traffic belonging to other flows is $C-s$. Therefore, the maximum number of buckets in which the traffic of other flows can reach at least $T-s$ is $\lfloor\frac{C-s}{T-s}\rfloor$. The probability of a

flow passing the filter is bound by the probability of it being hashed into such a bucket.

$$p_s \leq \frac{\lfloor \frac{C-s}{T-s} \rfloor}{b} \leq \frac{C}{b(T-s)} = \frac{1}{k}\frac{T}{T-s} \blacksquare$$

Based on this lemma we can compute the probability that a small flow passes the parallel multistage filter.

Lemma 11 (1) *Assuming the hash functions used by different stages are independent, the probability of a flow of size s passing a parallel multistage filter is bound by* $p_s \leq \left(\frac{1}{k}\frac{T}{T-s}\right)^d$.

Proof A flow passes the filter only if it passes all the stages. Since all stages are updated in the same way for the parallel filter, lemma 10 applies to all of them. Since the hash functions are independent, the probability of the flow passing all of the stages equals the product of the probabilities for every stage. ■

Now we can give the bound on the number of flows passing a multistage filter.

Theorem 12 (3) *The expected number of flows passing a parallel multistage filter is bound by*

$$E[n_{pass}] \leq max\left(\frac{b}{k-1}, n\left(\frac{n}{kn-b}\right)^d\right) + n\left(\frac{n}{kn-b}\right)^d \tag{B.1}$$

Proof Let s_i be the sequence of flow sizes present in the traffic mix. Let n_i the number of flows of size s_i. $h_i = \frac{n_i\, s_i}{C}$ is the share of the total traffic the flows of size s_i are responsible for. It is immediate that $\sum n_i = n$, and $\sum h_i = 1$. By Lemma 1 the expected number of flows of size s_i to pass the filter is $E[n_{i_{pass}}] = n_i p_{s_i} \leq n_i\ min(1, (\frac{1}{k}\frac{T}{T-s_i})^d)$. By the linearity of expectation we have $E[n_{pass}] = \sum E[n_{i_{pass}}]$.

To be able to bound $E[n_{pass}]$, we will divide flows in 3 groups by size. The largest flows are the ones we cannot bound p_{s_i} for. These are the ones with $s_i > T\frac{k-1}{k}$. The smallest flows are the ones below the average flow size of $\frac{C}{n}$. For these $p_{s_i} \leq p_{\frac{C}{n}}$. The number of below average flows is bound by n. For all these flows taken together $E[n_{small_{pass}}] \leq np_{\frac{C}{n}}$. The middle group is that of flows between $\frac{C}{n}$ and $T\frac{k-1}{k}$.

$$
\begin{aligned}
E[n_{pass}] &= \sum E[n_{i_{pass}}] = \sum_{s_i > T\frac{k-1}{k}} E[n_{i_{pass}}] + \sum_{\frac{C}{n} \le s_i \le T\frac{k-1}{k}} E[n_{i_{pass}}] + \sum_{s_i < \frac{C}{n}} E[n_{i_{pass}}] \\
&\le \sum_{s_i > T\frac{k-1}{k}} \frac{h_i C}{s_i} + \sum_{\frac{C}{n} \le s_i \le T\frac{k-1}{k}} \frac{h_i C}{s_i} \left(\frac{1}{k}\frac{T}{T-s_i}\right)^d + n\left(\frac{1}{k}\frac{T}{T-\frac{C}{n}}\right)^d \\
&\le C\left(\sum_{s_i > T\frac{k-1}{k}} h_i \frac{1}{T\frac{k-1}{k}} + \sum_{\frac{C}{n} \le s_i \le T\frac{k-1}{k}} h_i \frac{1}{s_i}\left(\frac{1}{k}\frac{T}{T-s_i}\right)^d\right) + n\left(\frac{1}{k}\frac{T}{T-\frac{C}{n}}\right)^d
\end{aligned}
$$

It's maybe easier to follow how the proof proceeds from here on if we assume that we have an adversary that tries to arrange the flows on purpose so that the largest number of possible flows passes the filter. But this adversary has a budget limited by the total amount of traffic it can send (the h_is have to add up to (at most) one because he cannot send more than the link bandwidth). We can see that the adversary can achieve the highest number by spending the traffic it allocates to flows above $T\frac{k-1}{k}$ to flows exactly at $T\frac{k-1}{k}$. This is equivalent to noticing that substituting all flows from this group with a number of flows of size $T\frac{k-1}{k}$ that generate the same amount of traffic is guaranteed to not decrease the lower bound for $E[n_{pass}]$. The next step is based on the observation that the number of flows passing the filter is maximized when the adversary chooses the size of flows in the middle group that maximizes the number of flows expected to pass the filter for a given amount of total traffic.

$$
\begin{aligned}
E[n_{pass}] &\le C\left(\sum_{\frac{C}{n} \le s_i \le T\frac{k-1}{k}} h_i \frac{1}{s_i}\left(\frac{1}{k}\frac{T}{T-s_i}\right)^d\right) + n\left(\frac{1}{k}\frac{T}{T-\frac{C}{n}}\right)^d \\
&\le C \max_{\frac{C}{n} \le s_i \le T\frac{k-1}{k}} \frac{1}{s_i}\left(\frac{1}{k}\frac{T}{T-s_i}\right)^d + n\left(\frac{1}{k}\frac{T}{T-\frac{C}{n}}\right)^d
\end{aligned}
$$

Next we determine the maximum of the function $f(x) = \frac{1}{x}(\frac{1}{T-x})^d$ on the domain $[\frac{C}{n}, T\frac{k-1}{k}]$.

$$
f'(x) = -\frac{1}{x^2}\left(\frac{1}{T-x}\right)^d + \frac{1}{x}\frac{d}{(T-x)^{d+1}} = \frac{1}{x}\frac{1}{(T-x)^d}\left(-\frac{1}{x} + \frac{d}{T-x}\right)
$$

Within $[\frac{C}{n}, T\frac{k-1}{k}]$ $f'(x) = 0$ for $x = \frac{T}{d+1}$ (if it is in the interval), $f'(x) < 0$ to the left of this value and $f'(x) > 0$ to the right of it. Therefore this represents a minimum for $f(x)$. Therefore the maximum of $f(x)$ will be obtained at one of the ends of the interval $C(\frac{T}{k})^d f(T\frac{k-1}{k}) = \frac{C}{T\frac{k-1}{k}} = \frac{b}{k-1}$ or $C(\frac{T}{k})^d f(\frac{C}{n}) = n(\frac{1}{k}\frac{T}{T-\frac{C}{n}})^d = n(\frac{nT}{knT-kC})^d = n(\frac{n}{kn-b})^d$. Substituting these values we obtain the bound. ■

For proving our high probability bounds, we use the following result from probability theory.

Lemma 13 *Assume we have a sequence of n independent events succeeding with probability p. The probability that the number of events succeeding i exceeds the expected value by more than λ is bound by*

$$Pr(i > np + \lambda) \leq e^{-\frac{\lambda^2}{2np+\frac{2}{3}\lambda}}$$

Corollary 13.1 *If we want to limit the probability of underestimation to p_{safe} for the experiment above we can bound i by*

$$i \leq \lfloor np - \frac{\ln(p_{safe})}{3} + \sqrt{\frac{\ln(p_{safe})^2}{9} - 2np\ln(p_{safe})} \rfloor$$

Proof By lemma 13 we have

$$e^{-\frac{\lambda^2}{2np+\frac{2}{3}\lambda}} \leq p_{safe}$$

We can determine λ by solving the resulting quadratic equation.

$$\lambda^2 + \frac{2\ln(p_{safe})}{3}\lambda + 2np\ln(p_{safe}) = 0$$

Since $\ln(p_{safe}) < 0$, the only positive solution is

$$\lambda = -\frac{\ln(p_{safe})}{3} + \sqrt{\frac{\ln(p_{safe})^2}{9} - 2np\ln(p_{safe})}$$

■

Theorem 14 *With probability p_{safe} the number of flows passing the parallel multistage filter is bound by*

$$n_{pass} \leq b - 1 + \lfloor n\left(\frac{1}{k-1}\right)^d + -\frac{\ln(p_{safe})}{3} + \sqrt{\frac{\ln(p_{safe})^2}{9} - 2n\left(\frac{1}{k-1}\right)^d \ln(p_{safe})} \rfloor$$

Proof We divide the flows into two groups: flows strictly above $\frac{C}{b}$ and flows below it. There are at most $b-1$ with $s > \frac{C}{b}$ and we assume that all of these pass. With Lemma 1 we bound the probability of passing for flows below $\frac{C}{b}$ by $(\frac{1}{k}\frac{T}{T-C/b})^d = (\frac{1}{k-1})^d$. The number of flows in this group is at most n. By applying corollary 13.1 we can bound the number of flows from this group passing the filter. Adding the numbers for the two groups gives us exactly the bound we need to prove. ■

For our algorithm strengthening this theorem we will divide the flows above $\frac{C}{b}$ into $k-2$ groups. The first group will contain all flows of $s > T\frac{k-2}{k}$ and we will assume that all of these pass. The jth group will contain flows of sizes between $T\frac{k-j-1}{k} < s \leq T\frac{k-j}{k}$. The last ($k-1$th) group will contain as in the case above, the flows with sizes below $\frac{C}{b} = \frac{T}{k}$.

Lemma 15 *The probability of an individual flow from group j passing the filter p_j and the number of flows in group j n_j will be bound by*

$$p_j \leq \left(\frac{1}{j}\right)^d$$

$$n_j \leq \begin{cases} \left\lfloor \frac{b}{k-j-1} \right\rfloor & \text{if } j < k-1 \\ n & \text{for the last group} \end{cases}$$

Proof For group 1 we have $p_1 \leq 1$, so it is a correct upper bound. For all the other groups we have an upper bound on the size of flows. Using Lemma 1 we see that no flow has a probability of passing larger than the probability for the largest permitted flow size. The bound for p_j is immediate.

For the last group the bound $n_{k-1} \leq n$ trivially holds because n is the total number of flows. All the other groups have a lower bound on the size of their flows. We know that the flows in a group cannot add up to more than the capacity of the link C. The bound on n_j is immediate. ■

Lemma 16 *If the distribution of flow sizes is Zipf, the number of flows in group j n_j will be bound by*

$$n_j \leq \begin{cases} \left\lfloor \frac{b}{(k-2)\ln(n+1)} \right\rfloor & \text{for the first group} \\ \left\lfloor \frac{b}{(k-j-1)\ln(n+1)} \right\rfloor - \left\lfloor \frac{b}{(k-j)\ln(2n+1)} \right\rfloor & \text{if}(j > 1 \text{ and } j < k-1) \\ n - \left\lfloor \frac{b}{(k-j)\ln(2n+1)} \right\rfloor & \text{for the last group} \end{cases}$$

Proof By applying Lemma 17, through simple manipulations, we obtain that the number of flows i larger than $T\frac{k-j}{k}$ is bound by

$$\left\lfloor \frac{b}{(k-j)\ln(2n+1)} \right\rfloor \leq i \leq \left\lfloor \frac{b}{(k-j)\ln(n+1)} \right\rfloor$$

Using these bounds, the lemma is immediate. ■

We can strengthen the bound from Theorem 14 by applying Corollary 13.1 to these groups. Each group will have a limit on the number of passing flows. For the first group this will be the number of flows. The probability of the total number of flows passing the filter exceeding the sum of these limits will be bound by the sum of the probabilities of individual groups exceeding their bounds. We divide p_{safe} evenly between the last $k-2$ groups.

There is one further optimization, we can apply in the distribution-free case. Since we derive the limits separately for the groups, it can happen that when we add up all the passing flows, we obtain traffic larger than C. We can discard the largest flows until the size of the passing flows is C. Figure B.1 gives the pseudocode of the resulting algorithm.

B.2 Analysis of algorithms for identifying large flows when flow sizes have a Zipf distribution

In this appendix we derive bounds on the number of memory entries required by sample and hold and multistage filters assuming the flow sizes have a Zipf distribution with parameter $\alpha = 1$.

B.2.1 Sample and hold with a Zipf distribution of flow sizes

Lemma 17 *If the sizes of flows have a Zipf distribution, we can bound from above and below the size of the i-th flow by* $\frac{C}{i\ln(2n+1)} \leq s_i \leq \frac{C}{i\ln(n+1)}$.

COMPUTEBOUND($psafe$)

$psafe = psafe/(k-2)$

for $j = 1$ to $k-1$

$p[j] = 1/(j^d)$

$n[j] = COMPUTEMAXFLOWCOUNT(j)$

$expectedpass[j] = n[j] * p[j]$

$smallest[j] = T * (k-1-j)/k$

if $(j == 1)$

$worstcasepass[j] = n[j]$

else

$lambda[j] = COMPUTELAMBDA(expectedpass[j], psafe)$

$worstcasepass[j] = \lfloor min(expectedpass[j] + lambda[j], n[j]) \rfloor$

endif

endfor

$passingflows = 0$

$passingtraffic = 0$

for $j = k-1$ to 1

$newtraffic = worstcasepass[j] * smallest[j]$

if$(newtraffic + passingtraffic > C)$

$worstcasepass[j] = (C - passingtraffic)/smallest[j]$

$newtraffic = worstcasepass[j] * smallest[j]$

endif

$passingflows+ = worstcasepass[j]$

$passingtraffic+ = newtraffic$

endfor

return $passingflows$

Figure B.1: Algorithm for computing a tight high probability bound on the number of flows passing a parallel filter

Proof The sizes of flows are $s_i = \gamma \frac{1}{i}$. We know that $\sum_{i=1}^{n} s_i = C$.

$$\begin{array}{lll} \int_i^{i+1} \frac{1}{x} dx & \leq \frac{1}{i} & \leq \int_{i-0.5}^{i+0.5} \frac{1}{x} dx \\ \gamma \int_1^{n+1} \frac{1}{x} dx & \leq \sum_{i=1}^{n} s_i & \leq \gamma \int_{0.5}^{n+0.5} \frac{1}{x} dx \\ \gamma \ln(n+1) & \leq C & \leq \gamma(\ln(n+0.5) - \ln(0.5)) = \gamma \ln(2n+1) \end{array}$$

■

Corollary 17.1 *If the sizes of flows have a Zipf distribution, the number of flows above a certain threshold T is at most $\lfloor \frac{C}{T \ \ln(n+1)} \rfloor$.*

Corollary 17.2 *If the sizes of flows have a Zipf distribution, the number of flows above a certain threshold T is at least $\lfloor \frac{C}{T \ln(2n+1)} \rfloor$.*

Lemma 18 *The first x flows represent at least a fraction of $\frac{\ln(x+1)}{\ln(2n+1)}$ of the total traffic.*

Proof

$$\sum_{i=1}^{x} s_i \geq \gamma \ln(x+1) \geq \frac{C}{\ln(2n+1)} \ln(x+1)$$

■

Based on this, we can compute that the total traffic of the first j flows is at least $C \frac{\ln(j+1)}{\ln(2n+1)}$. The exepected number of entries needed will be $j + Cp(1 - \frac{\ln(j+1)}{\ln(2n+1)})$. By differentiating, we see that we obtain the lowest value for the number of entries by choosing $j = \frac{Cp}{\ln(2n+1)} - 1$.[1] By substituting we obtain the number of entries we need in the flow memory $Cp(1 - \frac{\ln(Cp) - \ln(\ln(2n+1)) - 1}{\ln(2n+1)}) - 1$. The standard deviation of the number of sampled packets belonging to flows smaller than the jth is $\sqrt{Cp(1-p)(1 - \frac{\ln(j+1)}{\ln(2n+1)})}$. Applying Chebyshev's inequality we obtain that the probability that the number of entries required be larger than $Cp(1 - \frac{\ln(Cp) - \ln(\ln(2n+1)) - 1}{\ln(2n+1)}) - 1 + k\sqrt{Cp(1-p)(1 - \frac{\ln(j+1)}{\ln(2n+1)})}$ is less than $\frac{1}{k^2}$.

[1] Actually we have to choose either the integer just below or the one just above this value, but we ignore this detail for simplicity.

B.2.2 Multistage filters with a Zipf distribution of flow sizes

For proving theorem 4 we first need a helper lemma.

Lemma 19 *For any $\gamma > 0$ and $\gamma + 1.5 \leq i_0 < n$ we have*

$$\sum_{i=i_0}^{n} \left(\frac{1}{1-\frac{\gamma}{i}}\right)^d < n + 1 - i_0 + d\gamma \left(\ln(n+1) + \left(\frac{1}{1-\frac{\gamma}{i_0-0.5}}\right)^{d-1}\right)$$

Proof

$$\begin{aligned}
\sum_{i=i_0}^{n} \left(\frac{1}{1-\frac{\gamma}{i}}\right)^d &= \sum_{i=i_0}^{n} \frac{i^d}{(i-\gamma)^d} = \sum_{j=i_0-\gamma}^{n-\gamma} \frac{(j+\gamma)^d}{j^d} = \sum_{j=i_0-\gamma}^{n-\gamma} \sum_{m=0}^{d} \frac{\binom{d}{m} j^{d-m}\gamma^m}{j^d} \\
&= \sum_{m=0}^{d} \binom{d}{m} \gamma^m \sum_{j=i_0-\gamma}^{n-\gamma} j^{-m} \leq \sum_{m=0}^{d} \binom{d}{m} (-\gamma)^m \int_{j=i_0-\gamma-0.5}^{n-\gamma+0.5} j^{-m} dj \\
&= n + 1 - i_0 + \gamma d \int_{j=i_0-\gamma-0.5}^{n-\gamma+0.5} \frac{1}{j} dj + \sum_{m=2}^{d} \binom{d}{m} \gamma^m \int_{j=i_0-\gamma-0.5}^{n-\gamma+0.5} j^{-m} dj \\
&= n + 1 - i_0 + d\gamma \ln\left(\frac{n-\gamma+0.5}{i_0-\gamma-0.5}\right) \\
&\quad + \sum_{m=2}^{d} \binom{d}{m} m\gamma^m ((i_0-\gamma-0.5)^{-m+1} - (n-\gamma+0.5)^{-m+1})
\end{aligned}$$

$$\begin{aligned}
\sum_{m=2}^{d} \binom{d}{m} m\gamma^{m-1}(a^{-m+1} - b^{-m+1}) &= d \sum_{m=2}^{d} \binom{d-1}{m-1} \left(\left(\frac{\gamma}{a}\right)^{m-1} - \left(\frac{\gamma}{b}\right)^{m-1}\right) \\
&= d \sum_{r=1}^{d-1} \binom{d-1}{r} \left(\left(\frac{\gamma}{a}\right)^{r} - \left(\frac{\gamma}{b}\right)^{r}\right) \\
&= d \sum_{r=0}^{d-1} \binom{d-1}{r} \left(\frac{\gamma}{a}\right)^{r} - d \sum_{r=0}^{d-1} \binom{d-1}{r} \left(\frac{\gamma}{b}\right)^{r} \\
&= d\left(\left(1+\frac{\gamma}{a}\right)^{d-1} - \left(1+\frac{\gamma}{b}\right)^{d-1}\right) \\
&= d\left(\left(\frac{a+\gamma}{a}\right)^{d-1} - \left(\frac{b+\gamma}{b}\right)^{d-1}\right)
\end{aligned}$$

By combining these two results we immediately obtain

$$\begin{aligned}\sum_{i=i_0}^{n}\left(\frac{1}{1-\frac{\gamma}{i}}\right)^d &\leq n+1-i_0+d\gamma\left(\ln\left(\frac{n-\gamma+0.5}{i_0-\gamma-0.5}\right)+\left(\frac{i_0-0.5}{i_0-\gamma-0.5}\right)^{d-1}\right.\\ &\quad\left.-\left(\frac{n+0.5}{n-\gamma+0.5}\right)^{d-1}\right)\\ &< n+1-i_0+d\gamma\left(\ln(n+1)+\left(\frac{1}{1-\frac{\gamma}{i_0-0.5}}\right)^{d-1}\right)\end{aligned}$$

■

Theorem 20 (4) *If the flows sizes have a Zipf distribution, the expected number of flows passing a parallel multistage filter is bound by*

$$E[n_{pass}] \leq i_0+\frac{n}{k^d}+\frac{db}{k^{d+1}}+\frac{db\ln(n+1)^{d-2}}{k^2\left(k\ln(n+1)-\frac{b}{i_0-0.5}\right)^{d-1}} \tag{B.2}$$

where $i_0=\lceil max(1.5+\frac{b}{k\ln(n+1)},\frac{b}{\ln(2n+1)(k-1)})\rceil$.

Proof We divide the flows into two groups. As in the general case, for the larger ones we will assume they will pass. For the smaller ones we will use lemma 19 to bound the expected value of the number of flows passing. Before deciding where to separate the two groups we will give the general formula for the second one using lemma 1 (i_0 is the rank of the largest flow in this group).

$$\begin{aligned}E[n_{small_{pass}}] &= \sum_{i=i_0}^{n}p_{s_i}\leq\sum_{i=i_0}^{n}\left(\frac{1}{k}\frac{T}{T-s_i}\right)^d\leq\frac{1}{k^d}\sum_{i=i_0}^{n}\left(\frac{T}{T-\frac{C}{i\ln(n+1)}}\right)^d\\ &= \frac{1}{k^d}\sum_{i=i_0}^{n}\left(\frac{1}{1-\frac{\frac{b}{k\ln(n+1)}}{i}}\right)^d\end{aligned}$$

For lemma 19 to apply we need $i_0\geq 1.5+\frac{b}{k\ln(n+1)}$. To be able to bound the probability of these flows passing the filter, by lemma 10 we need $s_{i_0}\leq T\frac{k-1}{k}$. Through lemma 17 we obtain $i_0\geq\frac{\gamma k}{T(k-1)}\geq\frac{b}{\ln(2n+1)(k-1)}$. To satisfy both inequalities we set i_0 to $\lceil max(1.5+\frac{b}{k\ln(n+1)},\frac{b}{\ln(2n+1)(k-1)})\rceil$.

$$
\begin{aligned}
E[n_{pass}] &= \sum_{i=1}^{n} p_{s_i} = \sum_{i=1}^{i_0-1} p_{s_i} + \sum_{i=i_0}^{n} p_{s_i} \\
&\leq i_0 + \frac{n+1-i_0+\frac{db}{k\ln(n+1)}\left(\ln(n+1)+\frac{1}{\left(1-\frac{b}{k\ln(n+1)(i_0-0.5)}\right)^{d-1}}\right)}{k^d} \\
&\leq i_0 + \frac{n}{k^d} + \frac{db}{k^{d+1}} + \frac{db\ln(n+1)^{d-2}}{k^2\left(k\ln(n+1)-\frac{b}{i_0-0.5}\right)^{d-1}}
\end{aligned}
$$

■

Appendix C

Defining large flows with leaky buckets

In this appendinx we propose an alternate definition of large flows based on leaky buckets instead of measurement intervals. We also show how to adapt the multistage filters to this new definition and provide an analytical evaluation of the new scheme.

Defining large flows based on measurement intervals can lead to some unfairness. For example if a flow sends a burst of size slighlty larger than the threshold T within one measurement interval it is considered large. However, if the same burst spans two intervals, it's not. Even flows sending bursts of size almost $2T$ are not considered large if the bursts span measurement intervals a certain way. It can be argued that we should consider to be a large flow all flows that send more than T over any time interval no longer than a measurement interval. While this distinction is arguably not very important for the case of traffic measurement, it might matter for other applications.

We use a leaky bucket descriptor (also known as linearly bounded arrival process) to define large flows: a flow is large if during any time interval of size t it sends more than $r * t + u$ bytes of traffic. By properly choosing the parameters of the leaky bucket descriptor, we can ensure that all flows that send T bytes of traffic over a time interval no longer than a measurement interval are identified. We can adapt the multistage filters to this new definition by replacing the counters with "leaky buckets" and

instead of looking for counters above the threshold we look for buckets that violated the descriptor. We will first discuss how we can implement these efficiently at high speeds, and then give an analytical evaluation of the new algorithm.

C.1 Analytical evaluation of the parallel multistage filter using leaky buckets

Flows sending more than $r * t + u$ in any time interval of length t are large. For the example we used in section III.E.2 by setting r to 0.5 Mbytes/s and u to 0.5 Mbytes, we are guaranteed that flows that send 1 Mbyte during any second are labeled as large. This guarantees that we catch all flows that send more than 1 Mbyte during a measurement interval. We can conceptually describe the operation of the buckets as follows. Each bucket has a counter c initialized to 0. Every $\frac{1}{r}$ seconds this counter is decremented by 1 unless it is already 0 . When a packet of size s arrives, its size is added to the counter, but the value of the counter is not increased above u. If the counter is u, the incoming packet is considered to belong to a large flow. We also use the phrases "the bucket is in violation" and "the packet passes the bucket" to describe this situation. Section C.2 desctribes how this can be implemented efficiently. Actual implementations would probably use an approximation of this algorithm (e.g. they might decrement the leaky bucket less often), but we are not concerned with these details in our analysis. We use the notations below in our analysis.

- r the steady state data rate of the leaky bucket;
- u the burst size of the leaky bucket;
- C the data rate of the link (in bytes/second);
- k the stage strength: the ratio of r average data rate of the traffic through a bucket $k = \frac{r\ b}{C}$ (in our modified example above k is 5);
- τ the drain time for the leaky bucket $\tau = \frac{u}{r}$, for our example $\tau = 1$ second;
- c the counter of a certain leaky bucket (see below);

- a the number of "active" buckets in a stage (buckets with non-zero counters);
- A the active traffic in a particular stage defined as the sum of all counters;
- s the size of a packet or a sequence of packets;

We formalize the description of how the leaky buckets of the stages operate in the following two lemmas.

Lemma 21 *If $c_{initial}$ is the initial value of the counter of a bucket, after a time t where the bucket received no packets the value of the counter will be $c_{final} = max(0, c_{initial} - rt)$.*

Lemma 22 *If $c_{initial}$ is the initial value of the counter of a bucket when it receives a packet of size s, the value after the packet was processed is going to be $c_{final} = min(c_{initial} + s, u)$.*

Now we can prove a lemma that will help use prove we have no false negatives.

Lemma 23 *Let c be the value of the counter of a leaky bucket tracking a flow. Let c' be the counter of another bucket that counts all the packets of our flow and possibly packets of other flows. For any moment in time $c \leq c'$*

Proof By induction on time using as steps the moments when the packets are received (or any time interval shorter than the moment when the first packet is received).

Base case The buckets are exactly identical at the beginning of the interval $c = c$‘.

Inductive step Three things can happen: a packet belonging to the flow arrives, a packet not belonging to the flow arrives or no packets arrive for time t. In all three cases we will use the fact that by induction hypothesis, $c \leq c'$ in the beginning. If a packet of size s belonging to the flow arrives, by lemma 22 we have $c_{new} = min(c + s, u)$ and $c'_{new} = min(c' + s, u)$ therefore $c_{new} \leq c'_{new}$. If a packet of size s not belonging to the flow arrives, by lemma 22 we have $c_{new} = c$ and $c'_{new} = min(c' + s, u)$ therefore $c_{new} \leq c'_{new}$. If no packets arrive for time t, by lemma 21 at the end of the interval we have $c_{new} = max(0, c - rt)$ and $c'_{new} = max(0, c' - rt)$, therefore $c_{new} \leq c'_{new}$. ■

Corollary 23.1 *Let t be the moment in time when a certain flow exceeds the leaky bucket descriptor. The violation will be detected by the leaky bucket at time t no matter how many packets belonging to other flows hash to the same bucket.*

Theorem 24 *A parallel multistage filter will detect any flow exceeding the leaky bucket descriptor at latest when the flow does so.*

Proof By corollary 23.1, at all stages, the buckets the flow hashes to will detect the leaky bucket descriptor violation for the first packet of the flow that violates it, therefore this packet will pass the filter, causing the flow to be detected. ■

Just as in the case with the measurement intervals, we can have no false negatives and we want to bound the number of false positives. What we want to bound is the number of flows passing the filter during a certain time interval which gives us the peak rate at which new flows are added to the flow memory.

Lemma 25 *For any time interval t, if the counter of a bucket at the beginning was* $c_{initial}$ *and the traffic that hit the bucket during the interval is s, the final value of the counter is bound by* $c_{final} \leq max(0, c_{initial} - rt) + s$.

Proof By induction on time, using as steps the moments when the packets are received (or any time interval shorter than the moment when the first packet is received).

Base case At the beginning of the experiment, the time passed since the beginning of the experiment will be $t = 0$ and the sum of the sizes of the packets sent will be $s = 0$, therefore $c = c_{initial} = max(0, c_{initial} - rt) + s$.

Inductive step Two things can happen: a packet arrives or no packets arrive for time t'. In all cases we will use the fact that by induction hypothesis, $c \leq max(0, c_{initial} - rt) + s$ in the beginning where t is the time that passed since the beginning of the experiment and s is the sum of the sizes of the packets received so far. If a packet of size s' arrives, by lemma 22 we have $c_{new} = max(c + s', u) \leq c + s' \leq max(0, c_{initial} - rt) + s + s'$. If no packets arrive for time t', by lemma 21 at the end of the interval we have $c_{new} = max(0, c - rt') \leq max(0, max(0, c_{initial} - rt) + s - rt') \leq max(0, max(0, c_{initial} - rt) - rt') + s \leq max(0, max(0, c_{initial} - rt - rt')) + s = max(0, c_{initial} - r(t + t')) + s$. ■

Corollary 25.1 *The value of the counter of the bucket is not larger than the amount of traffic that the bucket received during the last τ.*

Lemma 26 *The active traffic in any stage is bound by $A \leq \frac{bu}{k}$.*

Proof By corollary 25.1, the size of each individual bucket will be bound by the traffic it received during the last τ, therefore A will be bound by the total traffic received during this interval which is bound by $C\tau = \frac{bu}{k}$. ■

Corollary 26.1 *At any moment in time the number of buckets in a stage a_x with $c \geq x$ is bound by $a_x \leq \lfloor \frac{bu}{kx} \rfloor$.*

We will bound the expected number of flows passing the filter during an interval of τ, the drain time for the leaky bucket. We cannot directly use Corollary 26.1 because a particular flow might pass the filter at any moment during the interval of τ.

Lemma 27 *The number of buckets of a stage with $c \geq x$ at any time during an interval of τ is bound by $a_x \leq \lfloor 2\frac{bu}{kx} \rfloor$.*

Proof By Lemma 26.1, the maximum number of buckets above x at the start of the interval is $\lfloor \frac{bu}{kx} \rfloor$ with the rest of the active traffic in other buckets. The best way an adversary could use the remaining active traffic at the beginning of the interval and the traffic sent during the interval is to fill buckets one by one. Since the amount of traffic sent during the interval is bound by $\frac{bu}{kx}$, by adding the number of buckets that were above c at the beginning to the ones that got filled up during the interval we obtain the bound of this lemma.■

Lemma 28 *For a flow that sends a total of s bytes during an interval τ and the preceding τ seconds, the probability that any of its packets pass the parallel multistage filter during the interval is bound by $p_s \leq \left(\frac{2}{k}\frac{u}{u-s}\right)^d$. If $s \leq u\frac{k-2}{k}$ this bound is below 1.*

Proof By Lemma 25.1, the size of the buckets it hashes to is bound by the traffic they received in the past τ seconds. This traffic is made up by traffic of the flow we are analyzing and traffic of other flows $c \leq s + s_{rest}$. The amount of traffic our flow sends during any window of τ seconds ending in the interval is bound by s. For the flow to pass

the filter, we need all buckets to pass the flow $s + s_{rest} \geq u$. By an argument similar to the one on Lemma 27, the number of buckets at each stage for which $s_{rest} \geq u - s$ at any moment during the interval is bound by $a_{u-s} \leq 2\frac{bu}{k(u-s)}$. Therefore the probability of passing any single stage is bound by $\frac{2u}{k(u-s)}$. This gives us the bound on the probability for a flow passing all of the stages as in the lemma. ■

Notice that this lemma is an upper bound, not the actual probability. It is even further from the real probability for the flow passing the filter than Lemma 1 because it assumes that for all stages s_{rest} reaches the right value exactly when the last packet of the flow is sent. This is quite unlikely in practice. Based on this lemma, we can give our final bound for the expected number of flows passing the filter during the interval τ.

Theorem 29 *The expected number of flows passing a multistage parallel filter during any interval of length τ is bound by*

$$E[n_{pass}] \geq max\left(\frac{2b}{k-2}, n\left(\frac{2n}{kn-2b}\right)^d\right) + n\left(\frac{2n}{kn-2b}\right)^d$$

Proof Let s_i be the sequence of flow sizes present in the traffic mix counting the traffic sent during the interval and the τ preceding seconds. Let n_i the number of flows of size s_i. $h_i = \frac{n_i\, s_i}{2C\tau}$ is the share of the total traffic that the flows of size s_i are responsible for. We have $\sum n_i = n$ (n is defined as the number of flows active during the interval, not the interval and the τ preceding seconds), and $\sum h_i = 1$. By lemma 28 the expected number of flows of size s_i to pass the filter is $E[n_{i_{pass}}] = n_i p_{s_i}$. By the linearity of expectation we have $E[n_{pass}] = \sum E[n_{i_{pass}}]$.

To be able to bound $E[n_{pass}]$, we will divide flows in 3 groups by size. The largest flows are the ones we cannot bound p_{s_i} for. These are the ones with $s_i > u\frac{k-2}{k}$. For these $E[n_{i_{pass}}] \leq n_i = \frac{h_i 2C\tau}{s_i} < \frac{h_i 2C\tau}{u\frac{k-2}{k}}$, therefore substituting them with a number of flows of size $u\frac{k-2}{k}$ that generate the same amount of traffic is guaranteed to not decrease the lower bound for $E[n_{pass}]$. The smallest flows are the ones below the average flow size of $\frac{2C\tau}{n}$. For these $p_{s_i} \leq p_{\frac{2C\tau}{n}}$. The number of below average flows is bound by n. For all these flows taken together $E[n_{small_{pass}}] \leq np_{\frac{2C\tau}{n}}$.

$$
\begin{aligned}
E[n_{pass}] &= \sum E[n_{i_{pass}}] = \sum_{s_i > u\frac{k-2}{k}} E[n_{i_{pass}}] + \sum_{\frac{2C\tau}{n} \le s_i \le u\frac{k-2}{k}} E[n_{i_{pass}}] + \sum_{s_i < \frac{2C\tau}{n}} E[n_{i_{pass}}] \\
&\le \sum_{s_i > u\frac{k-2}{k}} \frac{h_i 2C\tau}{s_i} + \sum_{\frac{2C\tau}{n} \le s_i \le u\frac{k-2}{k}} \frac{h_i 2C\tau}{s_i} \left(\frac{2}{k}\frac{u}{u-s_i}\right)^d + n\left(\frac{2}{k}\frac{u}{u-\frac{2C\tau}{n}}\right)^d \\
&\le 2C\tau \left(\sum_{s_i > u\frac{k-2}{k}} h_i \frac{1}{u\frac{k-2}{k}} + \sum_{\frac{2C\tau}{n} \le s_i \le u\frac{k-2}{k}} h_i \frac{1}{s_i}\left(\frac{2}{k}\frac{u}{u-s_i}\right)^d \right) \\
&\quad + n\left(\frac{2}{k}\frac{nu}{nu-2C\tau}\right)^d \\
&\le \frac{2ub}{k} \max_{\frac{2C\tau}{n} \le s_i \le u\frac{k-2}{k}} \left(\frac{1}{s_i}\left(\frac{2}{k}\frac{u}{u-s_i}\right)^d \right) + n\left(\frac{2n}{kn-2b}\right)^d
\end{aligned}
$$

As we saw in the proof of theorem 3, the maximum is reached at one of the ends of the interval. By substituting these values we obtain the bound. ■

If we compute the number for our example, we obtain a bound of 5,202.7 flows, which is much higher than the 121.2 theorem 3 gave. But is the comparison fair? Are the problems solved in the two cases equivalent? In the analysis with measurement intervals the number of flows that could violate the threshold during the measurement interval is 100. What is this number in our case? We can have 199 flows that keep their buckets at 0.5 Mbytes before our interval starts and they send one single small packet during the interval. These packets are all in violation and they should be detected. After this, we can have 198 other flows sending bursts of slightly more than 0.5 Mbytes so that they violate their leaky bucket descriptor. These flows should also all be passed by the filter if it is to avoid false negatives. Therefore we have a traffic pattern that requires at least 397 flows to be detected during the interval. If we proportionately increase the number of buckets at each stage from 1000 to $b = 4000$, theorem 29 gives us a bound of 454.6 which is approximately 4 times the bound of theorem 3. As with that result, we expect that in practice the number of flows passing will be much smaller.

C.2 Implementing multistage filters with leaky buckets

A naive implementation of the leaky buckets that make up the stages would keep decrementing the counters by 1 every $1/r$ seconds. This needs a lot of memory accesses and is not necessary. We think of the counters as numbers that move between 0 and u and what matters to the algorithm is where the counters are within this interval. Instead of decrementing all the counters every $1/r$ seconds by one, we can move the interval: we will have a virtual 0 and a virtual u that get *incremented* every $1/r$ seconds. Since we can keep these values in two registers, incrementing them often does not pose problems. With these new definitions, the counters themselves work the following way: when a new packet hashes to the counter, we first check if the value of the counter is below the virtual 0. If so we update it to 0. We add the size of the packet to the counter and if it is above the virtual u, we decrement it to virtual u. Finally, if the counter reached the virtual u we declare that the bucket is in violation. While this might sound complex, it needs no more memory accesses than the counters of filters operating with measurement intervals. With this implementation, we need to worry about overflows. We can implement the operations in such a way that when the virtual 0 and virtual u overflow, comparisons and arithmetic operations still work correctly. However, after an overflow an old counter that received no packets can seem to have a very large value instead of a very small one. To solve this problem we can use a background process that periodically updates to virtual 0 all the counters below it. Improvements to the basic parallel filter such as shielding and conservative update easily generalize to our filter using leaky buckets.

Appendix D

Measurements of algorithms for idenifying large flows

D.1 Measurements of sample and hold

In this appendix we present a detailed discussion of our measurements of the performance on sample and hold and its optimizations. Some of the less important results were omitted (all results are in [EV02]), but all results are discussed. We first compare the measured performance of the sample and hold algorithm to the values predicted by our analysis. Next we measure the improvement introduced by preserving entries across measurement intervals. In the last subsection we measure the effect of early removal and determine a good value for the early removal threshold.

We use 3 measures for the performance of the sample and hold algorithm: the average percentage of large flows that were not identified (false negatives), the average error of the traffic estimates for the large flows, and the maximum number of locations used in the flow memory.

D.1.1 Comparing the behavior of the basic algorithm to the analytic results

Our first set of experiments looks at the effect of oversampling on the performance of sample and hold. We configure sample and hold to measure the flows above

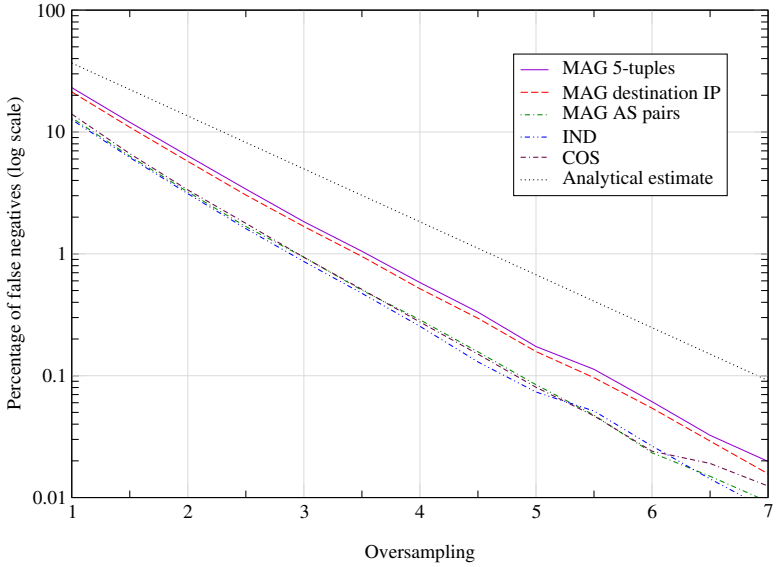


Figure D.1: Percentage of false negatives as the oversampling changes

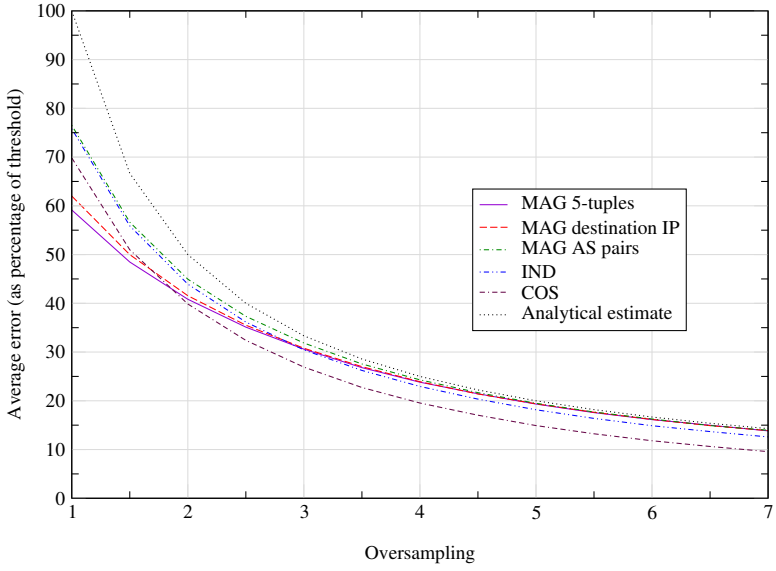


Figure D.2: Average error in the traffic estimates for large flows (without corrections to account for missed packets)

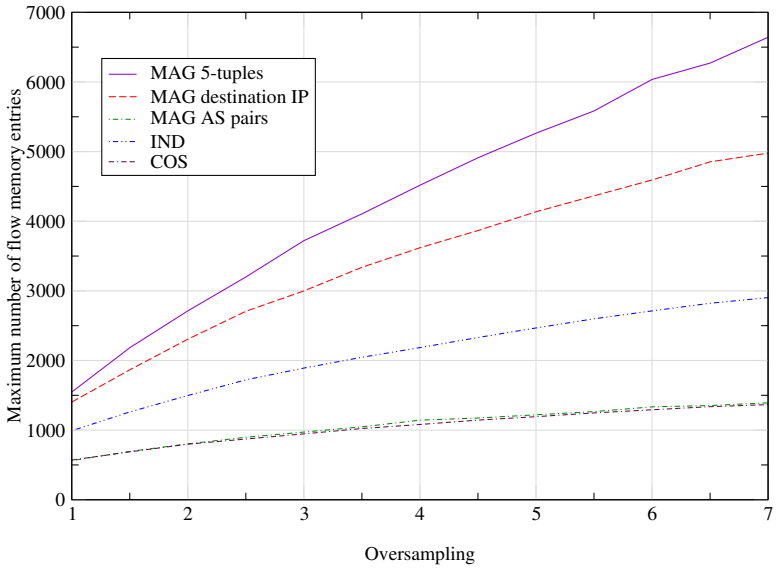


Figure D.3: Maximum number of flow memory entries used

0.01% of the link bandwidth and vary the oversampling factor from 1 to 7 (a probability of between 37% and less than 0.1% of missing a flow at the threshold). We perform each experiment for the traces MAG, IND and COS and for the trace MAG we use all 3 flow definitions (see Section III.H for a description of the traces). For each configuration, we perform 50 runs with different random functions for choosing the sampled packets. Figure D.1 shows the percentage of false negatives (the Y axis is logarithmic). We also plot the probability of false negatives predicted by our conservative analysis. The measurement results are considerably better than predicted by the analysis. The reason is that the analysis assumes that the size of the large flow is exactly equal to the threshold while most of the large flows are much above the threshold making them much more likely to be identified. The configurations with many large flows close to the threshold have false negative ratios closest to the results of our conservative analysis. The results confirm that the probability of false negatives decreases exponentially as the oversampling increases. Figure D.2 shows the average error in the estimate of the size of an identified large flow. The measured error is slightly below the error predicted by

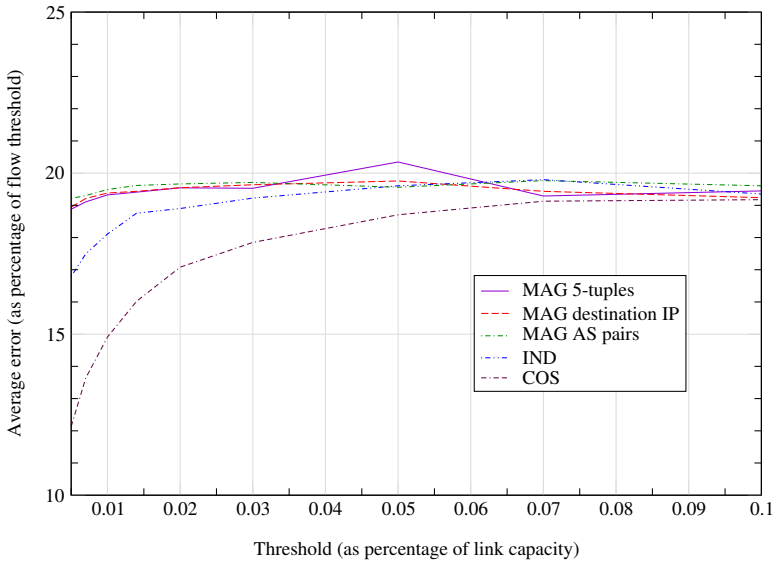


Figure D.4: Average error in the traffic estimates for large flows (without corrections to account for missed packets)

the analysis. The results confirm that the average error of the estimates is proportional to the inverse of the oversampling. Figure D.3 shows the maximum over the 900 measurement intervals for the number of entries of flow memory used. The measurement results are more than an order of magnitude lower than the bound from Section III.E.1. There are two main reasons: the links are lightly loaded (between 13% and 27%), and many of the sampled packets belonging to large flows do not create new entries in the flow memory. The results also show that the number of entries used depends on the number of active flows, and the dependence is stronger as the sampling probability (the oversampling) increases.

The next set of experiments looks at how the choice of the threshold influences the performance of the sample and hold algorithm. We run the algorithm with a fixed oversampling factor of 5 for thresholds between 0.005% and 0.1% of the link bandwidth. The most interesting result is Figure D.4 showing the average error in the estimate of the size of an identified large flow. As expected, the actual values are usually slightly below

Trace+flow definition	False neg. ratio min/median/max	Average error ratio min/median/max	Memory ratio min/median/max
MAG 5-tuple	0%/95.2%/200%	77.4%/90.6%/92.6%	64.5%/69.3%/81.0%
MAG destination IP	0%/90.5%/100%	79.9%/90.4%/98.2%	66.0%/72.3%/87.3%
MAG AS pairs	50%/92.4%/100%	78.7%/88.9%/93.2%	74.8%/80.5%/91.8%
IND 5-tuple	55.6%/92.0%/160%	81.4%/89.5%/96.2%	73.6%/80.5%/91.4%
COS 5-tuple	0%/84.5%/104%	77.5%/85.0%/92.3%	78.6%/82.6%/92.5%

Table D.1: Various measures of performance when using an early removal threshold of 15% of the threshold as compared with the values without early removal

the expected error of 20% of the threshold. The only significant deviations are for the traces IND and especially COS at very small values of the threshold. The explanation is that the threshold approaches the size of a large packet (e.g., a threshold of 0.005% on an OC3 (COS) corresponds to 4860, bytes while the size of most packets of the large flows is 1500 bytes). Our analysis assumes that we sample at the byte level. In practice, if a certain packet gets sampled, all its bytes are counted, including the ones before the byte that was sampled.

D.1.2 The effect of preserving entries

For all traces, we performed two sets of experiments: with fixed threshold and varying oversampling and with fixed oversampling and varying the threshold. The improvement introduced by preserving entries is not influenced much by the oversampling, but it is influenced considerably by the choice of the threshold. We conjecture that this happens because the magnitude of the improvement depends on the distribution of the durations for large flows and this changes as we change the threshold because the mix of large flows changes. Preserving entries reduces the probability of false negatives by 50% - 85%. It reduces the average error by 70% - 95%. The reduction is strongest when large flows are long lived. Preserving entries increases memory usage by 40% - 70%. The increase is smallest when large flows make up a larger share of the traffic.

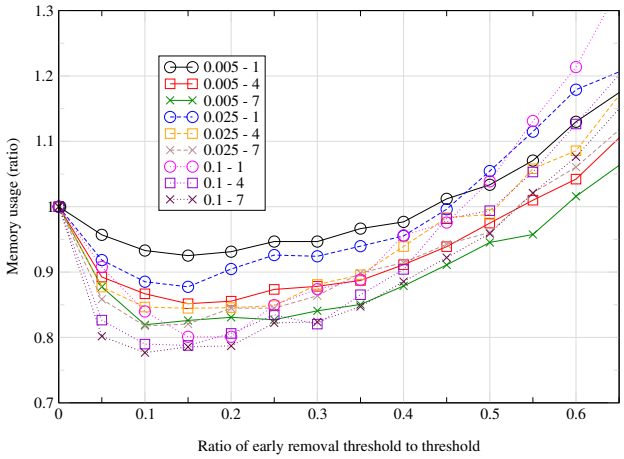


Figure D.5: Effect of early removal on memory usage

D.1.3 The effect of early removal

To measure the effect of early removal, we used 9 configurations with oversampling of 1, 4 and 7 and with thresholds of 0.005% 0.025% and 0.1% of the link bandwidth. For each of these configurations, we measure a range of values for the early removal threshold. We adjust the oversampling such that the probability of missing a flow at the threshold stays the same as without early removal (see Section III.E.1 for details). The point of these experiments is to obtain the value for the early removal threshold that results in the smallest possible memory usage. We performed 50 runs on the COS trace for each configuration. The measurements show that the probability of false negatives decreases slightly as the early removal threshold increases. This confirms that we compensate correctly for the large flows that might be removed early (through the increase the oversampling). Results also confirm our expectation that the average error decreases roughly linearly as the early removal threshold increases (due to the compensatory increase in oversampling). Figure D.5 shows that there is an optimal value for the early removal threshold (as far as memory usage is concerned) around 15% of the threshold. From the results we can also conclude that the larger the threshold the more memory we save but the less we gain in accuracy with early removal. Also the larger the

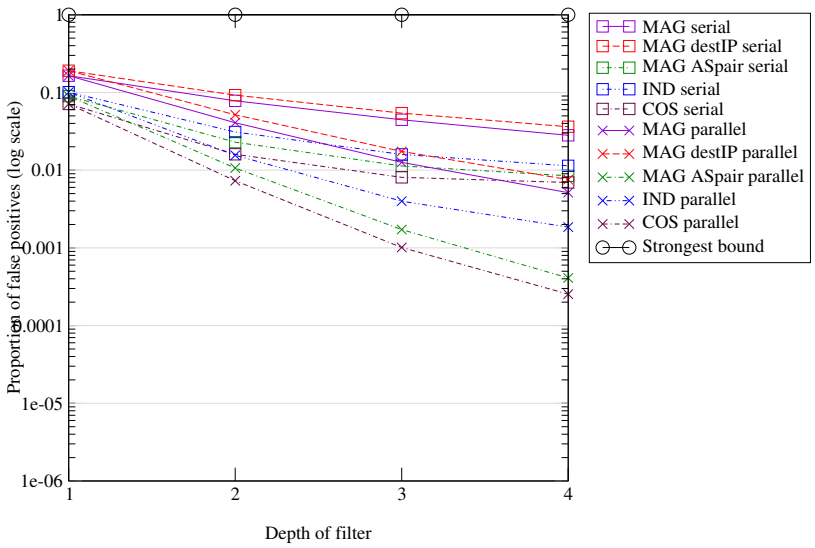


Figure D.6: Actual performance for a stage strength of k=1

oversampling, the more we gain in accuracy and memory.

The results for other traces and other flow definitions have very similar trends, but the actual improvements achieved for various metrics are sometimes different. Table D.1 has the minimum, median and maximum values (among the 9 configurations) for the 3 metrics of interest when using an early removal threshold of 15% of the threshold. All values are reported as ratios to the values obtained without early removal.

D.2 Measurements of Multistage filters

We first compare the performance of serial and parallel multistage filters to the bound of Theorem 3. Next, we measure the benefits of conservative update. In the last subsection we measure the combined effect of preserving entries and shielding.

D.2.1 Comparing the behavior of basic filters to the analytic results

First we compare the number of false positives for serial and parallel filters with the bound of Theorem 3. While the number of flow memory locations used might seem

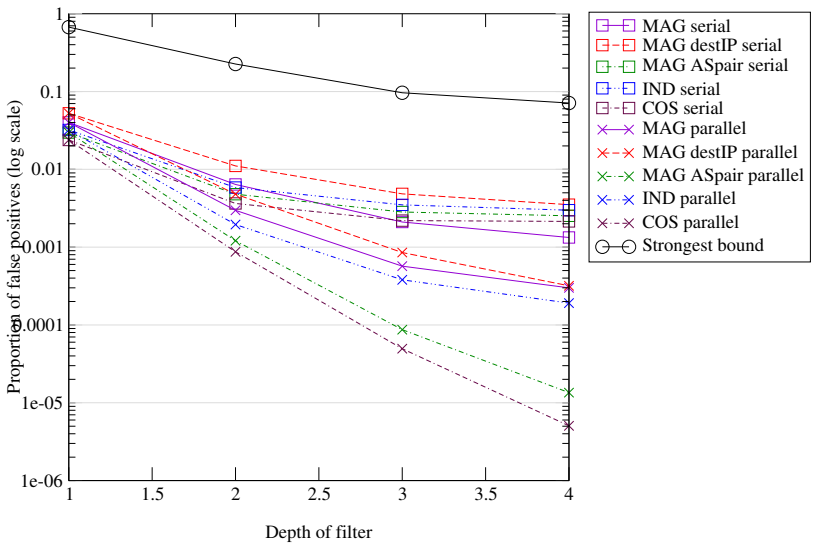


Figure D.7: Actual performance for a stage strength of k=3

like a more meaningful measure of the performance of the algorithm, we use the number of false positives because for strong filters. The number of entries is dominated by the entries of the actual large flows, making it harder to distinguish changes of even an order of magnitude in the number of entries occupied by false positives. To make it easier to compare results from different traces and different flow definitions (therefore different numbers of active flows), we actually report the percentage of false positives, not their number. Another important detail is that we express the threshold as a percentage of the maximum traffic, not as a percentage of the link capacity. While actual implementations do not know the traffic in advance, this choice of thresholds gives us information about how the filters would behave under extreme conditions (i.e., a fully loaded link). In this first set of experiments, we fix the threshold to a 4096th of the maximum traffic and vary the stage strength from 1 to 4 and the depth of the filter from 1 to 4 (the number of counters used by the filter is between 4K and 64K). For each configuration we measure 10 runs with different random hash functions. Figure D.6 and D.7 present the results of our measurements for stage strengths of 1 and 3 (all results are in [EV02]). We also represent the strongest bound we obtain from Theorem 3 for the configurations

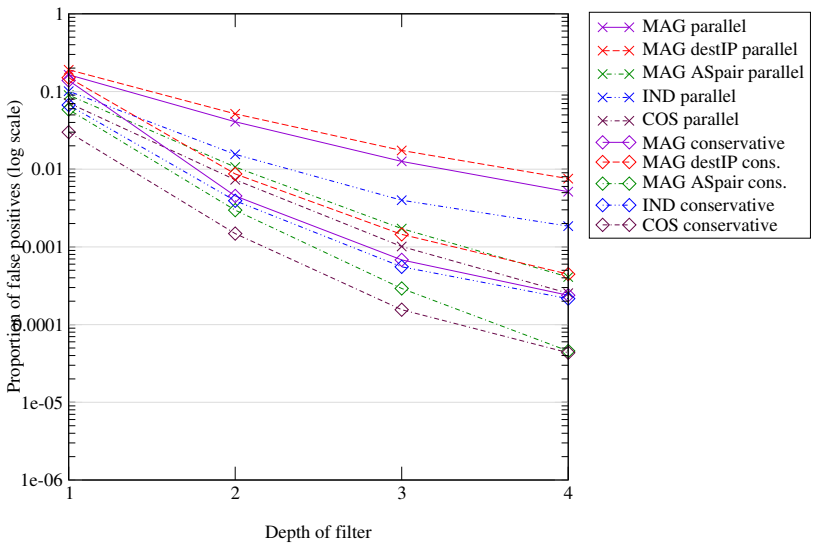


Figure D.8: Conservative update for a stage strength of k=1

we measure. Note that the y axis is logarithmic.

The results show that the filtering is in general at least an order of magnitude stronger than the bound. Parallel filters are stronger than serial filters with the same configuration. The difference grows from nothing in the degenerate case of a single stage to up to two orders of magnitude for four stages. The actual filtering also depends on the trace and flow definition. We can see that the actual filtering is strongest for the traces and flow definitions for which the large flows strongly dominate the traffic. We can also see that the actual filtering follows the straight lines that denotes exponential improvement with the numbering of stages. For some configurations, after a certain point, the filtering doesn't improve as fast anymore. Our explanation is that the false positives are dominated by a few flows close to threshold. Since the parallel filters clearly outperform the serial ones we use them in all of our subsequent experiments.

D.2.2 The effect of conservative update

Our next set of experiments evaluates the effect of conservative update. We run experiments with filter depths from 1 to 4. For each configuration we measure 10 runs

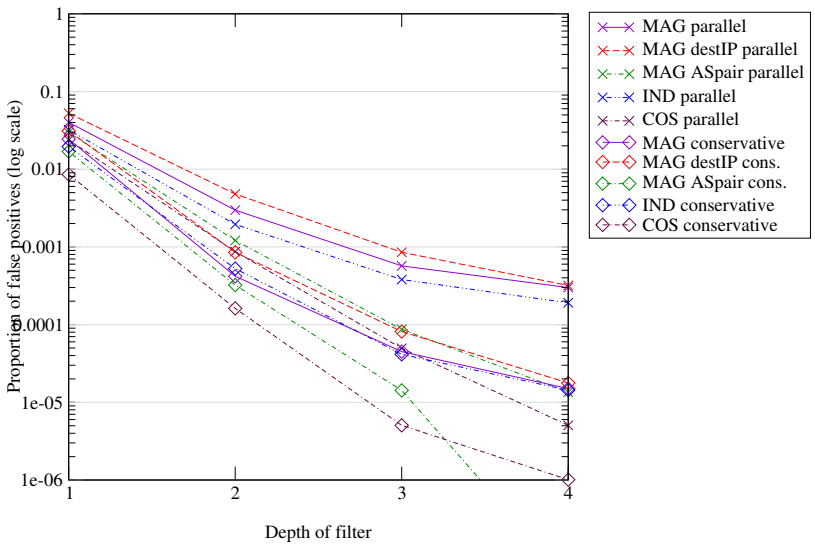


Figure D.9: Conservative update for a stage strength of k=3

with different random hash functions. For brevity we only present in figures D.8 and D.9 the results for stage strengths of 1 and 3. The improvement introduced by conservative update grows to more than an order of magnitude as the number of stages increases. For the configuration with 4 stages of strength 3 we obtained no false positives when running on the MAG trace with flows defined by AS pairs and that is why the plotted line "falls off" so abruptly. Since by extrapolating the curve we would expect to find approximately 1 false positive, we consider that this data point does not invalidate our conclusions.

D.2.3 The effect of preserving entries and shielding

Our next set of experiments evaluates the combined effect of preserving entries and shielding. We run experiments with filter depths from 1 to 4 and stage strengths of 0.5 and 2. We measure the largest number of entries of flow memory used and the average error of the estimates. When computing the maximum memory requirement we ignored the first two measurement intervals in each experiment because the effect of shielding is fully visible only from the third measurement interval on.

Trace + flow definition	Error when preserving entries
MAG 5-tuple	19.12% - 26.24%
MAG destination IP	23.50% - 29.17%
MAG AS pairs	16.44% - 17.21%
IND 5-tuple	23.46% - 26.00%
COS 5-tuple	30.97% - 31.18%

Table D.2: Average error when preserving entries as percentage of the average error in the base case

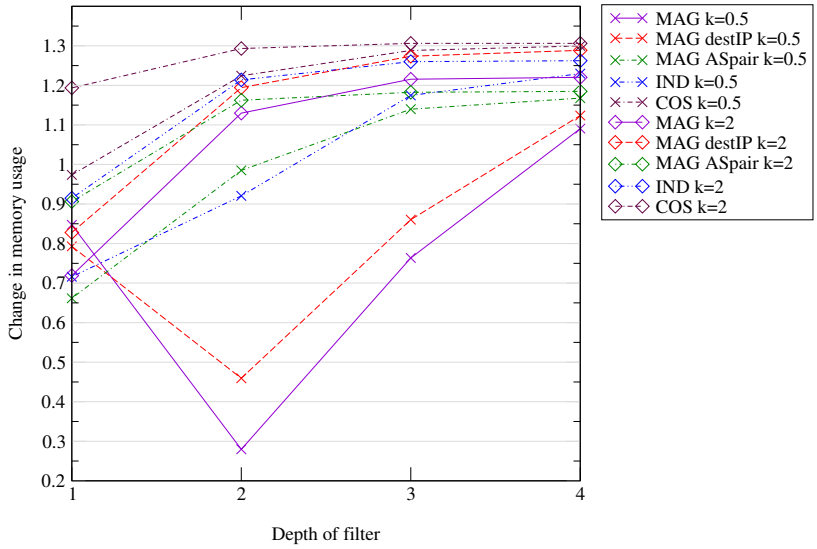


Figure D.10: Change in memory usage due to preserving entries and shielding

The improvement in the average error does not depend much on the filter configuration. Table D.2 shows the results for each trace and flow definition. Usually for the weak filters (few, weak stages), the reduction in the average error is slightly larger than for the strong ones.

There are two conflicting effects of preserving entries on the memory requirements. On one hand, by preserving entries we increase the number of entries used. On the other hand, shielding increases the strength of the filter (see Section III.E.2 for details), which leads to a decrease in the number of false positives. Figure D.10 shows

how memory usage is influenced by preserving entries. The first effect predominates for strong filters, leading to an increase in memory usage by up to 30%. The second one predominates for weak filters leading to a decrease by as much as 70%.

Appendix E

Multiresolution bitmap details

E.1 Analysis of average error for the virtual and multiresolution bitmaps

We first determine the variance of the estimate for the number of flows hashing to the virtual bitmap $VAR[\widehat{m}]$. The number of flows hashing to the bitmap m is a random variable distributed binomially. For this analysis we will assume that there are two independent hash functions, one that decides which flows map to the virtual bitmap, and one that decides how those get mapped to individual bits.[1] Therefore the analysis of the direct bitmap from Sections IV.D.1 and IV.D.2 describes the conditional distribution of the random variable $\widehat{m}$ given the value of m.

$$\begin{aligned} E[\widehat{m}|m=i] &= i \\ VAR[\widehat{m}|m=i] &\approx b\left(e^{i/b}-i/b-1\right) \\ E[\widehat{m}^2|m=i] &= E[\widehat{m}|m=i]^2+VAR[\widehat{m}|m=i] \\ &\approx i^2+b\left(e^{i/b}-i/b-1\right) \end{aligned}$$

To obtain $VAR[\widehat{m}]$ we need $E[\widehat{m}]$ and $E[\widehat{m}^2]$.

[1] Any reasonable hash function will make the two processes independent, so in practice we can use a single hash function.

$$
\begin{aligned}
E[\widehat{m}] &= \sum_{i=0}^{n} P(m=i)E[\widehat{m}|m=i] \\
&= \sum_{i=0}^{n} P(m=i)i = E[m] = \alpha n \\
E[\widehat{m}^2] &= \sum_{i=0}^{n} P(m=i)E[\widehat{m}^2|m=i] \\
&\approx \sum_{i=0}^{n} P(m=i)\left(i^2 + b\left(e^{i/b} - i/b - 1\right)\right) \\
&= E[m^2] + bE\left[e^{m/b}\right] - bE[m/b] - bE[1] \\
&= E[m^2] + bE\left[e^{m/b}\right] - E[m] - b
\end{aligned}
$$

To compute $E\left[e^{m/b}\right]$, we use the Taylor expansion of $f(x) = e^{x/b}$ around the value $E[m]$.

$$
\begin{aligned}
e^{x/b} &= e^{E[m]/b} + \frac{1}{b}e^{E[m]/b}(x - E[m]) \\
&\quad + \frac{1}{2b^2}e^{E[m]/b}(x - E[m])^2 + \ldots \\
E\left[e^{m/b}\right] &\approx e^{E[m]/b} + \frac{1}{b}e^{E[m]/b}E[m - E[m]] \\
&= e^{E[m]/b} + \frac{1}{b}e^{E[m]/b}(E[m] - E[m]) = e^{E[m]/b}
\end{aligned}
$$

To see how far off we are with this approximation, we compute below the value of the expectation of the third term of the Taylor expansion, the first term we ignore. Its ratio to our approximate result gives some indication of how much we are off. For example, with $b = 200$, which is smaller than what we expect to be used in practice, and a flow density of $\rho = 8$, which is much above the flow densities virtual bitmaps would be expected to operate accurately in, we are off by less than 2%. We also note here that the contribution of further terms is even smaller because they have higher powers of b at the denominator.

$$
\begin{aligned}
E\left[\frac{1}{2b^2}e^{E[m]/b}(x-E[m])^2\right] &= \frac{1}{2b^2}e^{E[m]/b}E\left[(x-E[m])^2\right] \\
&= \frac{1}{2b^2}e^{E[m]/b}VAR[m] \\
&= \frac{1}{2b^2}e^{E[m]/b}n\alpha(1-\alpha) \\
&< \frac{1}{2b^2}e^{E[m]/b}n\alpha = \frac{E[m]}{2b^2}e^{E[m]/b} \\
\frac{E\left[\frac{1}{2b^2}e^{E[m]/b}(x-E[m])^2\right]}{e^{E[m]/b}} &< \frac{E[m]}{2b^2} = \frac{\rho}{2b}
\end{aligned}
$$

Now substituting our approximate value for $E[e^{m/b}]$ we get $E[\widehat{m}]$ which we use to compute $VAR[\widehat{m}]$.

$$
\begin{aligned}
E[\widehat{m}^2] &\approx E[m^2] + be^{E[m]/b} - E[m] - b \\
VAR[\widehat{m}] &= E[\widehat{m}^2] - E[\widehat{m}]^2 \\
&\approx E[m^2] - E[m]^2 + be^{E[m]/b} - E[m] - b \\
&= VAR[m] + be^{n\alpha/b} - n\alpha - b \\
&= n\alpha(1-\alpha) + be^{\rho} - n\alpha - b \\
&= be^{\rho} - n\alpha^2 - b < b\left(e^{\rho}-1\right)
\end{aligned}
$$

$$
\begin{aligned}
SD\left[\frac{\widehat{n}}{n}\right] &= \frac{SD[\widehat{m}]}{n\alpha} \lessapprox \frac{\sqrt{b\left(e^{\rho}-1\right)}}{n\alpha} = \frac{\sqrt{e^{\rho}-1}}{n\alpha/b\sqrt{b}} \\
&= \frac{\sqrt{e^{\rho}-1}}{\rho\sqrt{b}}
\end{aligned}
$$

The tightness of the bound depends on the term $n\alpha^2$. Since the whole variance $VAR[\widehat{m}]$ is at least $n\alpha(1-\alpha)$, we are off by a factor of at most $1-\alpha$, therefore if α is small (i.e., the virtual bitmap covers only a small portion of the hash space), this bound is tight, but as α approaches 1, the bound is not tight anymore. Indeed, for $\alpha = 1$ we have a direct bitmap whose accuracy is described by Equation IV.3, which can be significantly lower thanEquation IV.4 when the flow density is low.

The multiresolution bitmap bases its estimate of the number of active flows on its estimate $\widehat{m}$ for the number of flows hashing to the base component *and all finer*

ones. We use m_b for the number of flows hashing to the base component and m_f for the number of flows hashing to all finer components ($m = m_f + m_b$). For this analysis we do not treat the finer components individually, but replace them with a single component: we extend the next component after the base to cover all the finer components, thus its size increases from b to $bk/(k-1)$. This is equivalent to or-ing together the bits of all the finer components until they are all at the granularity of the first component after the base. The benefit of this simplification is that the result will not depend on the number of finer components, thus it will apply no matter which component we use as base. While it is intuitively obvious that or-ing bits together leads to loss of information, and thus increases the variance of $\widehat{m_f}$, we can also prove it. What we need to show is that given a number of flows m_f that hash to the t finer components, the collision error we get when adding the estimates of these components is no bigger than the collision error of the component we get by collapsing them together. With $\rho = m_f/(b(k-1)/k)$, using the assumption that the collision errors are independent, we can write out the condition the two variances have to satisfy.

$$\begin{aligned}\frac{bk}{k-1}\left(e^{\rho} - \rho - 1\right) \quad \geq \quad & \sum_{i=0}^{t-2} b\left(e^{\rho/k^i} - \frac{\rho}{k^i} - 1\right) + \\ & \frac{bk}{k-1}\left(e^{\rho/k^{t-1}} - \frac{\rho}{k^{t-1}} - 1\right)\end{aligned}$$

The terms linear in ρ cancel out and the rest can be proven by summing inequalities of the type $1/k^i(e^{\rho} - 1) \geq e^{\rho/k^i} - 1$, which hold for all positive values of k, i and ρ based on simple calculus.

As with the virtual bitmap, we assume that the hash function deciding which component a flow gets mapped to and the two hash functions deciding to which of the bits of the component the flow is mapped are independent. While the sampling errors of $\widehat{m_b}$ and $\widehat{m_f}$ are correlated, the correlation is negative and its value is small when the sampling factor is large, so we ignore it. Because of the independence of mapping flows to bits, the collision errors are uncorrelated.

$$
\begin{aligned}
VAR[\widehat{m_b}] &\lessapprox b\,(e^{\rho_b}-1) = b\,(e^{\rho}-1) \\
VAR[\widehat{m_f}] &\lessapprox \frac{bk}{k-1}\,(e^{\rho_f}-1) = \frac{bk}{k-1}\left(e^{\rho/k}-1\right) \\
VAR[\widehat{m}] &= VAR[\widehat{m_b}] + VAR[\widehat{m_f}] + COV[\widehat{m_b},\widehat{m_f}] \\
&< VAR[\widehat{m_b}] + VAR[\widehat{m_f}] \\
&\lessapprox b\left(e^{\rho}-1+\frac{k}{k-1}\left(e^{\rho/k}-1\right)\right) \\
&= \frac{bk}{k-1}\left(\frac{k-1}{k}\,(e^{\rho}-1)+e^{\rho/k}-1\right) \\
SD\left[\frac{\widehat{n}}{n}\right] &= \frac{SD[\widehat{m}]}{n\alpha} = \frac{SD[\widehat{m}]}{\rho bk/(k-1)}
\end{aligned}
$$

$$
SD\left[\frac{\widehat{n}}{n}\right] \lessapprox \frac{\sqrt{\frac{k-1}{k}\,(e^{\rho}-1)+e^{\rho/k}-1}}{\rho\sqrt{\frac{bk}{k-1}}} \tag{E.1}
$$

Is the error introduced by collapsing all finer components into a single one acceptable? To answer this question we derived two formulas similar to Equation E.1: one that maintains one finer component and collapses all the rest (thus working with 3 bitmaps) and one maintaining two finer bitmaps and collapsing the rest (thus working with 4 bitmaps). We plugged in all three formulas into the algorithm for computing the sizes of the components of the multiresolution bitmaps. The more accurate (and more complicated) formulas always resulted in lower sizes for the bitmaps, but the differences were significant only for low values of k. Thus, the formula using 3 bitmaps reduces the bitmap size with respect to Equation E.1 by 3% for $k = 2$, and 1% for $k = 3$. Using 4 bitmaps reduces the bitmap size (with respect to the formula with 3 bitmaps) by less than 1% even for $k = 2$. As a tradeoff between accuracy and simplicity, we decided to use the formula derived based on 3 bitmaps, which is Equation IV.5 from Section IV.D.2.

E.2 Configuring the multiresolution bitmap

Figure E.1 presents the algorithm for numerically computing the optimal configuration for a multiresolution bitmap: the one that achieves the required average relative error ϵ up to the given number of flows N using the smallest amount of memory.

The algorithm checks four configurations. The first configuration (lines 4 to 7) assumes that the last two components are never used as base component for estimating the number of active flows. Equation IV.5 guarantees this configuration delivers the desired accuracy, but it turns out to be a wasteful one. The second configuration we consider (lines 8 to 12) also uses the last normal component as base (thus reducing the number of components needed by one) and increases the size of the last component until the accuracy of using the penultimate component (as given by Equation E.1) is good enough. This also turns out to be a wasteful configuration. The third configuration (lines 13 to 21) increases the size of the last component up to the point where it can take over as base component (based on Equation IV.4). Increasing the last component even further reduces the number of components by one more and this results sometimes in lower memory consumption. This is the fourth configuration we consider (lines 22 to 29). The algorithm for hardware configurations is very similar except that it takes care that $bk/(k-1)$ and b_{last} be powers of two.

Table IV.2 shows that $k = 2$ is asymptotically the best choice. There are some very rare cases when $k = 3$ gives a slightly smaller memory usage. This is because the number of components cannot be fractional and the components for $k = 3$ "fit better" to the given N and ϵ. The multiresolution bitmap is very easy to implement in hardware if k, $bk/(k-1)$ and b_{last} are powers of two. Under these constraints, sometimes the choice of $k = 4$ gives a smaller memory usage because the size b of the components it needs to achieve the desired average error ϵ "fits better" the powers of two. Therefore when configuring the algorithm for a hardware implementation that has these limitations it is best to check both values of $k = 2$ and $k = 4$. We found no set of parameters N,ϵ for which the hardware configuration with $k = 8$ used less memory than the smallest of $k = 2$ and $k = 4$.

E.3 Multi-resolution bitmap versus probabilistic counting

Even though it might seem surprising at first, the data collected by a multiresolution bitmap with $k = 2$ and no stretching of the last component is isomorphic to the data collected by a probabilistic counting algorithm configured in a certain way

(assuming we have good hash functions). The probability of the incoming packet to hash to component i is $1/2^i$ for all components but the last, for which it is $1/2^{c-1}$. Each component but the last has $b/2$ bits.

The probability that the packet hashes to a given bit at component i is $1/2^i \cdot 2/b = 1/(b \cdot 2^{i-1})$ (this also holds for the last component). Therefore, for each i from 1 to $c-1$ we have $b/2$ bits that have a probability of $1/(b \cdot 2^{i-1})$ of "catching" the incoming packet, plus we have b bits that have the probability $1/(b \cdot 2^{c-1})$ of "catching" the incoming packet. All incoming packets map to exactly one bit.

Probabilistic counting of Flajolet and Martin uses $nmap$ bitmaps of size L. Each bitmap has a probability of $1/nmap$ of "catching" a random database record. Within each bitmap, bit i has a probability of $1/2^i$ of catching the record. The last bit acts as a "catch-all" for all numbers of consecutive zeroes of L or more in the hash, so it has a probability of $1/2^{L-1}$ of catching the packet. Overall, for each i from 1 to $L-2$ we have $nmap$ bits that have a probability of $1/(nmap \cdot 2^i)$ of "catching" the record, plus we have $2 \cdot nmap$ bits that have a probability of $1/(nmap \cdot 2^{L-1})$ of "catching" the record.

We can see in Figure E.2 that when $b = 2 \cdot nmap$ and $c = L - 1$ the two algorithms have the same number of bits and the probability distribution of bits getting set is the same. Therefore, we conclude that the data collected by the two algorithms is equivalent. What is the difference then? The most important difference is the way the collected data is interpreted. As both analysis and experiments show this leads to our algorithm being more accurate when the number of flows is small. Another difference is that we have different rules for configuring the algorithm which, as experiments show, result in somewhat more accurate estimates when the number of flows is large.

E.4 Hardware implementation of multiresolution bitmap

While the analysis of the statistical behavior of multiresolution bitmap can look complicated, its implementation is simple. The most time-critical part of the algorithm that has to be performed for each packet is updating the bitmap. Computing the address of the bit to get updated is quite simple in hardware if three constraints are met: ratio

of the resolutions of neighboring components k, the size of the last component, and $bk/(k-1)$ need to be powers of two. Thus for the multiresolution bitmap example in Figure IV.2 ($b = 6$, $k = 4$) one can map the incoming packets to the proper sub-interval by computing a 7-bit hash function and using simple additional combinatorial logic. If the first two bits of the hash are not "11", the first 3 bits decide which of the sub-intervals in the coarsest component the packet maps to. Otherwise if the third and fourth bit are not 11 (but the first two are), bits from 3 to 5 decide which sub-interval in the intermediate component the packet maps to. If the first 4 bits are 1, bits from 5 to 7 map the packet to the appropriate subinterval in the finest component.

Figure E.3 presents a possible hardware implementation for the logic circuit that computes the address of the bit the current packet maps to. It is based on the operations described in the previous paragraph. The input to the circuit is the 7 bit hash function Hash[0..6] that is based on the flow ID of the packet. Its output is a 5 bit address Addr[0..4] of the bit that will be set. The leftmost bit in the multiresolution bitmap of Figure IV.2 has an address of 0 and the rightmost has an address of 19. The "select resolution" block selects the component with the appropriate resolution based on the first 4 bits of the hash. If the coarsest component is used, the output Res[0,1] will have a value of 0, and if the finest component is used, its value will be 2. This block can be implemented with a few gates. The "X 6" block multiplies this value by 6 because there are 6 bits in each component. Since this block performs multiplication with a constant value, it can be implemented using an adder and a shift register. The "Select offset" block selects which of the bits within the hash to be used to find the offset Offset[0..2] of the bit within the component. It can be implemented with a 16:4 multiplexer. The final adder adds together the base and the offset to get the address of the bit.

COMPUTECONFIGURATION(N, ϵ)

1 $lowestMemorySoFar = \infty$

2 for $k = 2$ to 16

3 $(\rho_{min}, \rho_{max}) = (\rho_{min}[k], \rho_{max}[k])$

4 $b = \lceil errorCoefficient[k]/\epsilon^2 \rceil$

5 $c = 3 + \lceil log_k(N/(b\rho_{max})) \rceil$

6 $b_{last} = \lceil errorCoefficient(k)/\epsilon^2 \ k/(k-1) \rceil$

7 CONSIDERCONFIGURATION(c, k, b, b_{last})

8 $c = c - 1$

9 while $\frac{\sqrt{b(e^{\rho_{max}}-1)+b_{last}\left(e^{\rho_{max}/(k-1)b/b_{last}}-1\right)}}{b\rho_{max}k/(k-1)} > \epsilon \mid \frac{\sqrt{b(e^{\rho_{min}}-1)+b_{last}\left(e^{\rho_{min}/(k-1)b/b_{last}}-1\right)}}{b\rho_{min}k/(k-1)} > \epsilon$

10 $b_{last} = b_{last} + 1$

11 endwhile

12 CONSIDERCONFIGURATION(c, k, b, b_{last})

13 while $\frac{\sqrt{b_{last}\left(e^{\rho_{max}/(k-1)b/b_{last}}-1\right)}}{b\rho_{max}/(k-1)} > \epsilon$

14 $b_{last} = b_{last} + 1$

15 endwhile

16 $\rho_{maxlast} = 1.6$

17 while $\frac{\sqrt{\left(e^{\rho_{maxlast}+0.00001}-1\right)}}{(\rho_{maxlast}+0.00001)\sqrt{b_{last}}} < \epsilon$

18 $\rho_{maxlast} = \rho_{maxlast} + 0.00001$

19 endwhile

20 $c = 1 + \lceil log_k(N/(b_{last}\rho_{maxlast})) \rceil$

21 CONSIDERCONFIGURATION(c, k, b, b_{last})

22 while $c - 1 < \lceil log_k(N/(b_{last}\rho_{maxlast})) \rceil$

23 $b_{last} = b_{last} + 1$

24 while $\frac{\sqrt{\left(e^{\rho_{maxlast}+0.00001}-1\right)}}{(\rho_{maxlast}+0.00001)\sqrt{b_{last}}} < \epsilon$

25 $\rho_{maxlast} = \rho_{maxlast} + 0.00001$

26 endwhile

27 endwhile

28 $c = 1 + \lceil log_k(N/(b_{last}\rho_{maxlast})) \rceil$

29 CONSIDERCONFIGURATION(c, k, b, b_{last})

30 endfor

31 return $(bestk, bestb, bestc, lowestMemorySoFar)$

CONSIDERCONFIGURATION(c, k, b, b_{last})

1 $Mem = b(c-1) + b_{last}$

2 if $Mem < lowestMemorySoFar$

3 $\quad (bestk, bestb, bestc, lowestMemorySoFar) = (k, b, c, Mem)$

4 endif

5 return

Figure E.1: The algorithm for numerically computing the best configuration for a multi-resolution bitmap considers four configurations for each value of k and returns the one using the least memory. For all software configurations we tested, the algorithm chose the third (line 21) or the fourth (line 29) configuration. For all hardware configurations it chose the third.

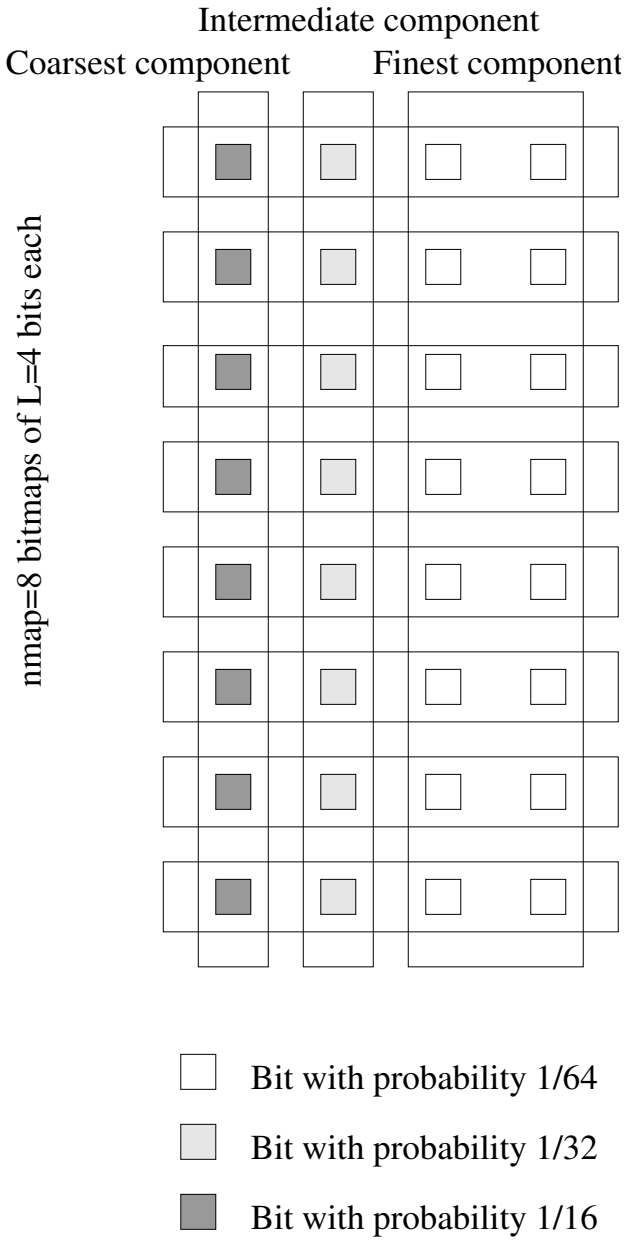


Figure E.2: Probabilistic counting groups the bits horizontally into bitmaps that contain bits with different probabilities of being set, while multiresolution bitmaps group bits vertically with bits with the same probability of being set in the same component

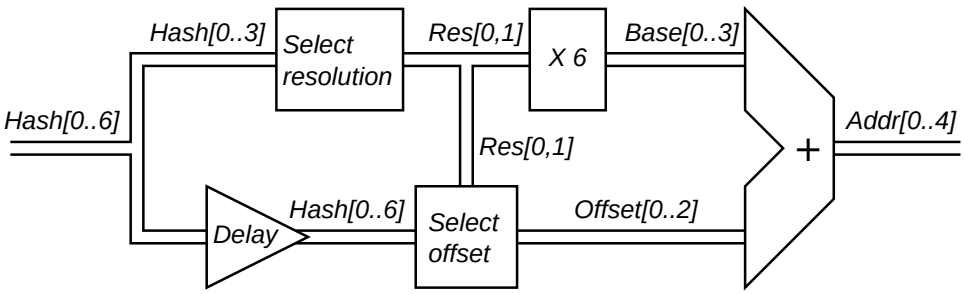


Figure E.3: Hardware for selecting the bit to be set

Appendix F

Measurements of the error of direct and virtual bitmap

In this appendix we present various experiments that evaluate how well direct bitmap and virtual bitmap behave. We first look at how well Equations IV.3 and IV.4 match the actual average error of direct and respectively virtual bitmaps. We use a bitmap of 1000 bits and measure the average relative error for flow densities ρ from 0.1 to 4 for the direct bitmap and from 0.4 to 4 for the virtual bitmap. Also, for the virtual bitmap we use two configurations with "sampling factors" of 10 and 100. For each of these setups, we generated synthetic traces based on packet headers from the MAG with the exact number of active flows required to reach the desired flow density. Each trace had 100 measurement intervals and we repeated each experiment 10 times with different hash functions, thus getting 1000 data points for each density for each configuration. Figures F.1 and F.2 show the relative errors we measured and how they compare to the values predicted by our analysis. We can see that observed behavior follows very closely the predictions of our theoretical analysis. As the flow density approaches 4, the measured error is very slightly above the predicted one. This can be explained with the effect of the terms of the Taylor series we ignored in the analysis.

We next look at what happens as we decrease the number of bits. Figures F.3 and F.4 show the results for bitmaps with 100 bits. The results are very similar except that by the time the flow density reaches 3, there are cases where the bitmap gets full

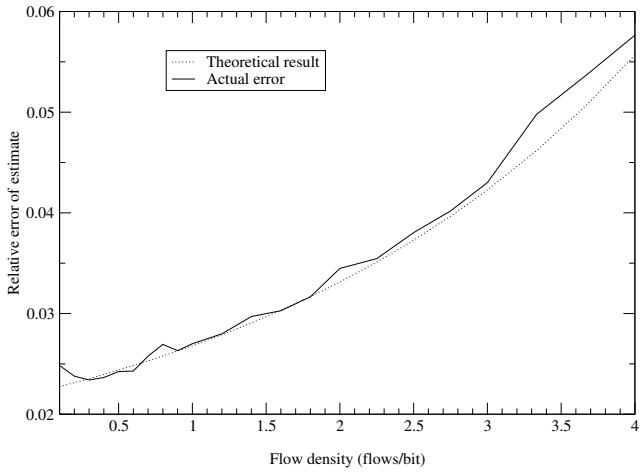


Figure ›its

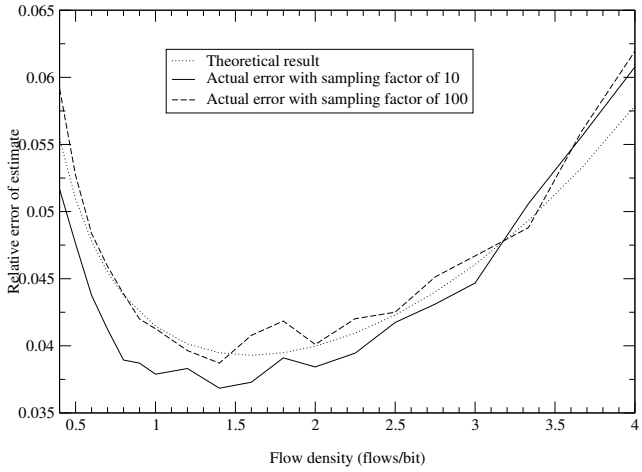


Figure F.2: Average relative error of a virtual bitmap with 1000 bits

and therefore cannot produce an estimate. We excluded from the results any data point for which at least one of the 10*100 measurement intervals produced a full bitmap. Reducing the bitmap size to 10 causes the direct bitmap to occasionally fill up as soon as we reach 10 active flows, and the virtual bitmap of size 10 can fill up for flow densities as low as 0.6.

We also measured the bias of the estimates. The results are presented in figures F.5 and F.6. The biases are statistically significant but much smaller than the average relative error of the respective configurations. The general tendency is to have

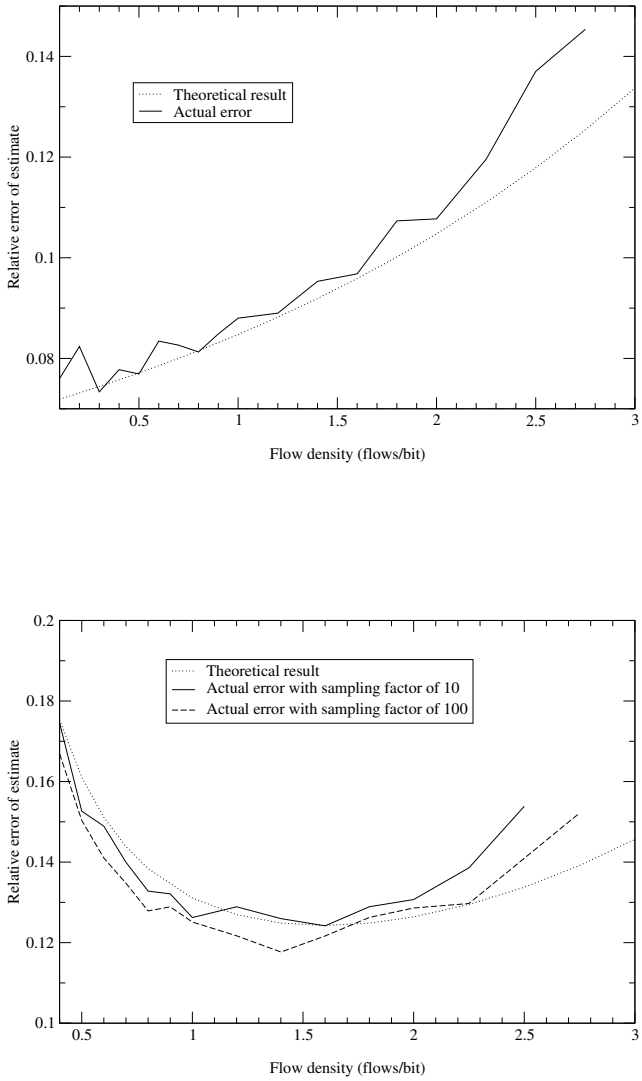


Figure F.4: Average relative error of a virtual bitmap with 100 bits

a negative bias at low flow densities and have it increase to positive values as the flow density increases. We observe that the configurations that have particularly bad and consistent negative bias, 100 bit virtual bitmap with a sampling factor of 100 and 1000 bit virtual bitmap with a sampling factor of 10, are those whose average errors were slightly above the theoretical analysis in figures F.2 and F.4. Those configurations used the same random hash functions. Our explanation for this slight anomaly is that some of those hash functions were more likely to produce collisions than the perfect hashes our analysis assumed, thus causing the estimates to be slightly lower on average. Use

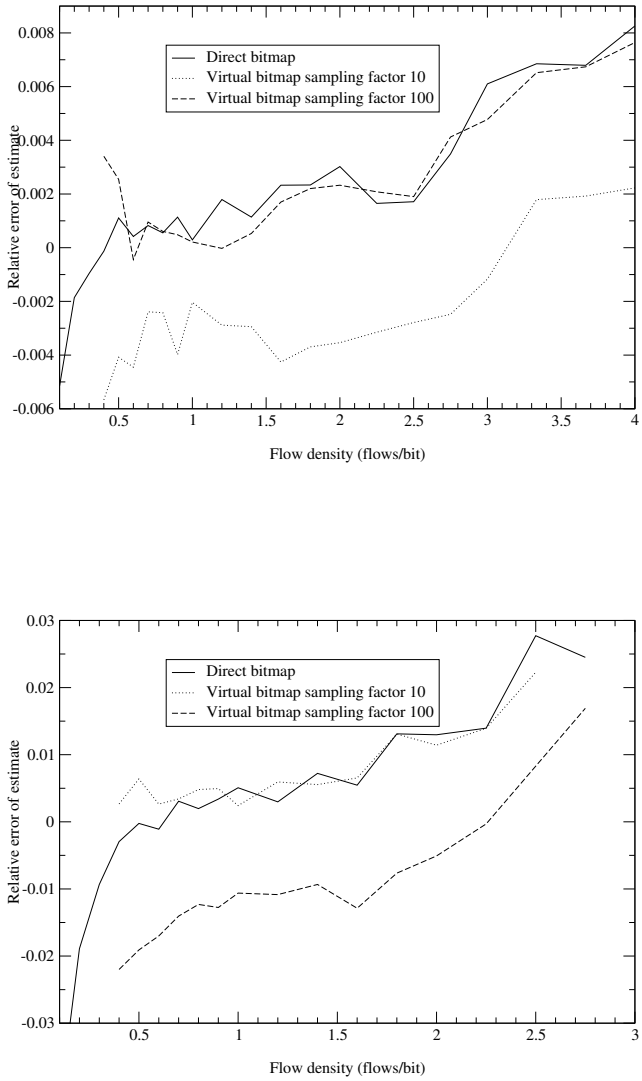


Figure F.6: Bias for the 100 bit bitmaps

of weaker CRC based hash functions instead of H3 lead to results significantly further from the theoretical analysis.

Appendix G

The size of reports using traffic clusters

We use measurements to answer a number of questions about the size of traffic reports based on traffic clusters. We compare theoretical bounds against actual results. We investigate the effect of the threshold, the length of the measurement intervals and traffic diversity on the size of the report. We also report running times for some of our algorithms. For these measurements we used our first trace from SD-NAP, and one from a backbone link. Both of these traces use a packet sampling of one in 400.

G.1 Comparison of actual report sizes and theoretical bounds for different threshold values

We measured the number of high volume traffic clusters (the uncompressed report) and the size of the compressed report for each of the 31 days of our first trace, using both definitions: defining traffic as the number of bytes and also defining it as the number of packets. We used thresholds of 0.5%, 1%, 2% 5% and 10% of the total traffic. The results are in Table G.1 together with the theoretical upper bounds on the sizes of the reports. The difference between the smallest and largest uncompressed report for the same configuration can be more than a factor of 100. A closer look at these extreme results explains these wide discrepancies. If there are a few very large TCP

H/T	Uncompressed size						
	Bound of Lemma 8	Byte report/10^3			Packet report/10^3		
		min	avg	max	min	avg	max
0.5%	2,433,600	96.4	177	280	75	132	219
1%	1,216,800	54.2	96.3	166	15	50	120
2%	608,400	26.9	50.9	104	2.8	21.1	54.7
5%	243,360	7.16	18.1	46.1	0.75	8.85	42.4
10%	121,680	0.32	9.87	34.0	0.16	3.16	31.5
H/T	Compressed size						
	Bound of Lemma 9	Byte reports			Packet reports		
		min	avg	max	min	avg	max
0.5%	93,600	74	334	576	195	514	664
1%	46,800	47	155	281	96	228	306
2%	23,400	25	72	126	47	105	126
5%	9,360	9	29	50	20	40	54
10%	4,680	6	14	20	10	17	22

Table G.1: Compressed reports are much shorter than uncompressed reports. Actual reports are orders of magnitude smaller than the theoretical bounds.

Number of intervals * length	Uncompressed report						Number of flows/10^3		
	Byte rep. size/10^3			Packet rep. size/10^3					
	min	avg	max	min	avg	max	min	avg	max
31 * 1 day	7.16	18.1	46.1	0.75	8.85	42.4	164	267	420
48 * 1 hour	0.75	16.2	41.6	0.62	8.87	41.7	9.89	20.6	195
48 * 5 min	1.06	22.0	38.9	0.65	15.8	36.6	0.83	12.9	24.9
Number of intervals * length	Compressed report						Number of flows/10^3		
	Byte report size			Packet report size					
	min	avg	max	min	avg	max	min	avg	max
31 * 1 day	9	29	50	20	40	54	164	267	420
48 * 1 hour	9	26	41	20	36	48	9.89	20.6	195
48 * 5 min	3	17	36	9	27	46	0.83	12.9	24.9

Table G.2: The size of compressed reports is slightly smaller for shorter measurement intervals. The number of active flows is an indication of the number records NetFlow would generate.

connections that dominate the traffic on a certain day, the uncompressed reports will be very large because there are very many more general clusters that include each of these fine-grained connections. Exactly the same type of traffic mix results in very small

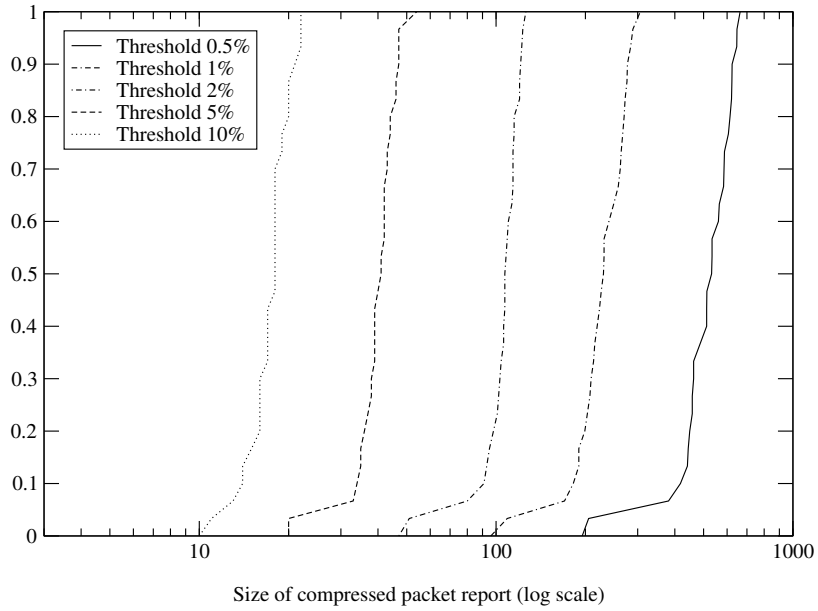


Figure G.1: The size of compressed traffic reports is proportional to the inverse of the threshold. For the same configuration, most of the traffic reports have very close sizes, except a few that are much smaller due to high volume individual connections that dominate the traffic mix (and push the threshold up).

Trace	Uncompressed report						Number of flows/10^3		
	Byte rep. size/10^3			Packet rep. size/10^3					
	min	avg	max	min	avg	max	min	avg	max
trace1	0.75	4.69	13.2	0.77	1.64	7.39	22.4	64.9	357
trace3/dir0	0.69	0.75	0.82	0.56	3.66	19.5	320	340	376
trace3/dir1	1.00	1.19	1.30	0.69	0.75	0.81	471	514	588
Trace	Compressed report						Number of flows/10^3		
	Byte report size			Packet report size					
	min	avg	max	min	avg	max	min	avg	max
trace1	28	36	41	32	37	41	22.4	64.9	357
trace3/dir0	34	38	43	34	40	45	320	340	376
trace3/dir1	33	40	45	42	45	50	471	514	588

Table G.3: The greater diversity of backbone traffic (trace3) did not lead to significant increases in the size of the traffic reports.

compressed reports, because once the individual large connections are reported, there is not much other traffic to report. Because large connections dominate more the byte breakdown than the packet breakdown, the compressed reports are somewhat smaller when we look at bytes and the uncompressed ones are smaller when we look at packets. For all configurations, the maximum number of high volume clusters is approximately within a factor of 10 of the bound of Lemma 8, but for all configurations there is more than a factor of 100 between the largest compressed reports reports and the bound of Lemma 9. This is encouraging, because the compressed reports are what we actually use.

G.2 The effect of the length of the measurement interval

We measure the effect of the length of the measurement interval on the size of the report using a threshold of 5% of the total traffic. We use our entire first trace to obtain 31 one day long measurement intervals. We use two days of the trace with very different traffic mixes, Sunday the 8th of December and Monday the 9th, to obtain 48 one hour measurement intervals. We use as five minute measurement intervals from minute 22 to minute 27 of each of these one hour intervals.

The results from Table G.2 suggest that the size of the compressed reports does not vary significantly with the length of the measurement intervals. Since shorter intervals have fewer flows, it is easier for the largest of them to dominate the traffic mix and this explains the slightly shorter compressed reports.

Note that except for the shortest intervals we considered (5 minutes) even the uncompressed reports are orders of magnitude smaller than the number of active flows, which is a rough indication of the number of NetFlow records that would be generated during the measurement interval. Since we computed the number of flows active in the sampled trace, this roughly matches the number of records sampled NetFlow would generate with the same sampling factor (1 in 400).

G.3 The effect of traffic diversity

We measure the effect of traffic diversity by comparing the size of the report using a threshold of 5% for the first trace and our third trace. Our third trace is very different: it is from an OC-48 backbone link. It is 8 hour long and it starts at 9:00 AM on the 14th of August 2002. This trace has both directions of the traffic. We divide the third trace into eight one hour measurement intervals. We use the same hours of the 16th of December (also a weekday) from our first trace.

The results from Table G.2 suggest that the size of the compressed reports does not vary significantly with the size of the location the traffic, but the more diverse backbone traffic leads to slightly longer reports. Even though the running time on the backbone trace was double the running time on the first trace, we interpret our results as an indication that our method might scale to backbone links too.

Bibliography

[AC01] J. Altman, and K. Chu, 2001. A proposal for a flexible service plan that is attractive to users and Internet service providers. In *IEEE Proceedings of the INFOCOM*.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD*, 1993.

[B70] B. Bloom, 1970. Space/time trade-offs in hash coding with allowable errors. In *Commun. ACM*. Vol. 13. 422–426.

[BELV+00] M. Burrows, U. Erlingson, S.-T. Leung, M. Vandevoorde, C. A. Waldspurger, and K. W. W. Weihl, 2000. Efficient and flexible value sampling. In *ASPLOS*.

[BJKST02] Ziv Bar-Yossef, T.S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan, Counting Distinct Elements in a Data Stream. RANDOM 2002

[BMR99] N. Brownlee, C. Mills, and G. Ruth, 1999. Traffic flow measurement: Architecture. RFC 2722.

[C] Coralreef - workload characterization. `http://www.caida.org/ analysis/ workload/`.

[CM03] S. Cohen, and Y. Matias, 2003. Spectral bloom filters. In *SIGMOD/PODS*.

[CMN98] S. Chaudhuri and R. Motwani and V. Narasayya, 1998. Random sampling for histogram construction: How much is enough? In *Proceedings of the ACM SIGMOD*.

[CN] Cisco NetFlow. `http://www.cisco.com /warp /public /732 /Tech /netflow`.

[CSN] Cisco sampled NetFlow. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120s/120s11/12s_sanf.htm`.

[CW00] Cisco offers wire-speed intrusion detection, December 2000. http://www.nwfusion.com/ reviews/ 2000/ 1218rev2.html.

[D] The DAG project `http://dag.cs.waikato.ac.nz`

[DG00] N. Duffield, and M. Grossglauser, 2000. Trajectory sampling for direct traffic observation. In *Proceedings of the ACM SIGCOMM*. 271–282.

[DKS89] A. Demers, S. Keshav, and S. Shenker, 1998. Design and Analysis of a Fair Queueing Algorithm. In *Proceedings of the ACM SIGCOMM*.

[DLT01] Nick Duffield, Carsten Lund, and Mikkel Thorup, 2001. Charging from sampled network usage. In *SIGCOMM Internet Measurement Workshop*.

[DLT02] Nick Duffield, Carsten Lund, and Mikkel Thorup. Properties and prediction of flow statistics from sampled packet streams. In *SIGCOMM Internet Measurement Workshop*, November 2002.

[DLT03] Nick Duffield, Carsten Lund and Mikkel Thorup, 2003, Estimating Flow Distributions from Sampled Flow Statistics. In *Proceedings of the ACM SIGCOMM*. 325–336.

[ESV03] Cristian Estan, Stefan Savage and George Varghese, 2003. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proceedings of the ACM SIGCOMM*.

[EVF03] Cristian Estan, George Varghese and Mike Fisk, 2003. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the Internet Measurement Conference*.

[EV02] Criatian Estan, and George Varghese, 2002. New directions in traffic measurement and accounting. Tech. Rep. 0699, UCSD CSE Department. Feb.

[EV02a] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM*, August 2002.

[EV03] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the Elephants, Ignoring the Mice. In ACM Transactions on Computer Systems, August 2003.

[F90] Philippe Flajolet On Adaptive Sampling COMPUTG: Computing (Archive for Informatics and Numerical Computation), Springer-Verlag", 43, 1990

[F98] Fyodor. Remote OS detection via TCP/IP stack fingerprinting. *Phrack*, (54), December 1998.

[FGLR+00] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, 2000. Deriving traffic demands for operational ip networks: Methodology and experience. In *Proceedings of the ACM SIGCOMM*. 257–270.

[FKSS01] W.-C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, 2001. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *IEEE Proceedings of the INFOCOM*.

[FM85] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, October 1985.

[FP99] W. Fang, and L. Peterson, 1999. Inter-as traffic patterns and their implications. In *Proceedings of IEEE GLOBECOM*.

[FSGMU98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, 1998. Computing iceberg queries efficiently. In *International Conference on Very Large Data Bases*. 307–317.

[GCBL+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, Hamid Pirahesh Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. J. Data Mining and Knowledge Discovery", 1997.

[GKMS01] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Quicksand: Quick summary and analysis of network data. DIMACS technical report, 2001.

[GM98] Pillip B. Gibbons, and Yossi Matias, 1998. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD*. 331–342.

[GM99] Phillip B. Gibbons and Yossi Matias, 1999, Synopsis Data Structures for Massive Data Sets, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on External Memory Algorithms and Visualization

[GM99a] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of the ACM SIGCOMM*, 1999.

[H01] John Huber, 2001. Design of an OC-192 flow monitoring chip. Class Project.

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proceeding of VLDB*, 1995.

[HTF01] T. Hastie, R. Tibshirani, and J. Friedman. The elements of statistical learning. Springer, 2001. pages 453-479.

[I] Ipmon - packet trace analysis. `http://ipmon.sprintlabs.com/ packstat/ packetoverview.php`.

[KHR02] Dina Katabi, Mark Handley, and Charlie Rhors. Congestion control for high bandwidth-delay product networks. In *Proceedings of the ACM SIGCOMM*, August 2002.

[KME03] Ken Keys, David Moore, and Cristian Estan. A Robust System for Accurate Real-time Summaries of Internet Traffic. CAIDA technical report `http://www.caida.org/outreach/papers/2003/flowest-tech/`, September 2003.

[KMKL+01] Ken Keys, David Moore, R. Koga, E. Lagache, M. Tesch, and K. Claffy. The architecture of coralreef: an internet traffic monitoring software suite. PAM2001, Workshop on Passive and Active Measurements, RIPE, 2001.

[KPS03] R. M. Karp, C. H. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in streams and bags. ACM TODS Volume 28, Number 1, March 2003.

[L79] C. E. Leiserson. Systolic priority queue. Caltech conference on VLSI, January 1979.

[LS94] P. Lavoie, Y. Savaria. A systolic architecture for fast stack sequential decoders. IEEE Transactions on Communications, 42(2-4),Feb-Apr 1994.

[LS98] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM*, pages 203–214, September 1998.

[M01] David Moore, 2001. Personal conversation. also see CAIDA analysis of code-red. `http://www.caida.org/ analysis/ security/ code-red/`.

[MBFI+01] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. ACM SIGCOMM Computer Communication Review, Volume 32 Issue 3, July 2002

[MPSS+02] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. IEEE Security and Privacy, July/August 2003.

[MR91] Keith McCloghrie and Marshall T. Rose RFC 1213 March 1991.

[MSR97] S.W. Moon, K. G. Shin, J. Rexford. Scalable hardware priority queue architectures for high-speed packet switches. In *Proceedings of Real-Time Applications Symposium*, June 1997.

[MTSBD02] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya and C. Diot. Traffic Matrix Estimation: Existing Techniques and New Directions, In *Proceedings of the ACM SIGCOMM*, 2002.

[MV95] J. Mackie-Masson, and H. Varian, 1995. *Public Access to the Internet.* MIT Press, Chapter Pricing the Internet.

[NSSCV03] S. Narayanasamy, T. Sherwood, S. Sair, B. Calder, and G. Varghese 2003. Catching accurate profiles in hardware. In *HPCA*.

[OR95] Tobias Oetiker and Dave Rand. MRTG: Multi Router Traffic Grapher `http://people.ee.ethz.ch/ oetiker/webtools/mrtg/` 1995.

[P97] Vern Paxson. Measurements and Analysis of End-to-End Internet Dynamics. Ph.D. Thesis, University of California, Berkeley. April 1997

[P99] Vern Paxson, Bro: a system for detecting network intruders in real-time Computer Networks (Amsterdam, Netherlands: 1999)

[P00] David Plonka. Flowscan: A network traffic flow reporting and visualization tool. In *LISA*, pages 305–317, December 2000.

[PH98] D. A. Patterson, and J. L. Hennessy, 1998. *Computer Organization and Design*, second ed. Morgan Kaufmann, 619.

[PPM01] Peter Phaal, Sonia Panchen and Neil McKee, RFC 3176: sFlow September 2001

[PPS01] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, 2001. Approximate fairness through differential dropping. Tech. rep., ACIRI.

[R99] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference*. USENIX, 1999.

[R00] L.G. Roberts, "Beyond Moore's law: Internet growth trends," IEEE Computer Mag., January 2000.

[RL] Riverstone Networks. LFAP: Lightweight flow accounting protocol. `http://www.riverstonenet.com/ technology/ accounting_for_profitability.shtml`.

[SBS01] S. Sastry, R. Bodik, and J. E. Smith, 2001. Rapid profiling via stratified sampling. In *28th. International Symposium on Computer Architecture*. 278–289.

[SCEH96] S. Shenker, D. Clark, D. Estrin, and S. Herzog, 1996. Pricing in computer networks: Reshaping the research agenda. In *ACM Computer Communications Review*. Vol. 26. 19–43.

[SEVS03] Sumeet Singh, Cristian Estan, George Varghese, Stefan Savage. The EarlyBird System for Real-time Detection of Unknown Worms. Technical report CS2003-0761, UCSD.

[SGDGL02] S. Saroiu, K. Gummadi, R. Dunn, S. Gribble, and H. Levy. An analysis of Internet content delivery systems. In *Proceedings of OSDI*, 2002.

[SHM01] Stuart Staniford, J. Hoagland, and J. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, (10), 2002.

[SKR01] Smitha, I. Kim, and A. L. N Reddy, 2001. Identifying long term high rate flows at a router. In *Proceedings of High Performance Computing*.

[SMW02] Neil Spring, Ratul Mahajan, David Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proceedings of the ACM SIGCOMM* 2002.

[SRS99] A. Shaikh, J. Rexford, and K.G Shin, 1999. Load-sensitive routing of long-lived IP flows. In *Proceedings of the ACM SIGCOMM*.

[SSZ98] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. In *Proceedings of the ACM SIGCOMM*, September 1998.

[T] tcpdump - dump traffic on a network `http://www.tcpdump.org`

[TMW97] K. Thomson, G. J. Miller, and R. Wilder. Wide-area traffic patterns and characteristics. In IEEE/ACM Transactions on Networking, pp 10-23, 1997.

[TR99] D. Tong, and A. L. N. Reddy, 1999. QoS enhancement with partial state. In *International Workshop on QOS*.

[V85] J.S. Vitter. Random sampling with a reservoir. ACM Trans. on Mathematical Software, 11(1), 1985.

[WVT90] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.

[YC96] Ming-Young You and Cheng-Shang Chang. Resampling for wireless access. In *Proceedings of IEEE PIMRC*, June 1996.