Binned Duration Flow Tracking and Symmetric Connection Detection

by

Bradley Whitehead

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Applied Science

Ottawa-Carleton Institute for
Electical and Computer Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
January 15, 2007

Your file Votre référence
ISBN: 978-0-494-23351-1
Our file Notre référence
ISBN: 978-0-494-23351-1

Abstract

Network measurement at 10+Gbps speeds imposes many restrictions on the resource consumption of the measurement application. In this thesis we describe two novel techniques that can perform per-flow flow monitoring on high-speed routers; Symmetric Connection Detection (SCD), and Binned Duration Flow Tracking (BDFT). Many network monitoring applications are only interested in TCP connections that become fully established, so other connection attempts, such as port scanning attempts, simply waste resources if not filtered. SCD filters out unsuccessful connection attempts by tracking the state of connection establishment for every flow observed. Using an upper bound of 32k bytes of RAM our SCD experimental results indicate 99+% accuracy with 900,000 active flows. Network operators require information such as the duration of a flow, or the distribution of flow durations to track today's quickly-changing network conditions. BDFT tracks the duration of every flow in a network, on a per-flow basis, and can report this information in real-time. Experimental results show that BDFT is over 99.8% accurate with only 352k bytes of memory.

Acknowledgments

Thanks to my supervisor Dr. Chung-Horng Lung, for his expertise, advice, and support during the evolution of this thesis. The general direction of the thesis was also shaped and supported during discussions with Peter Rabinovitch at Alcatel.

Support and funding for this research was provided by the Center for Communications and Information Technology, a division of Ontario Centers of Excellence, and Alcatel. The experimental results were generated from traces made available to us by CAIDA and NLANR.

Table of Contents

List of Tables

List of Figures

Nomenclature

BDFT - Binned Duration Flow Tracking

Bucket - In d-left hashing a *bucket* contains a fixed number of *cells.*

Cell - One location in a filter, this can be a bit, or a counter, or in d-left hashing can contain a whole *item*

DoS - Denial of Service (attack)

DDoS - Distributed Denial of Service (attack)

DPI - Deep Packet Inspection

DRAM - Dynamic Random Access Memory

FIN - Finish TCP packet flag (end connection)

ISP - Internet Service Provider

Item - A member of a set that is stored in one or more *cells* in a Bloom filter

NPU - Network Processing Unit

RST - Reset TCP packet flag (reset connection)

SCD - Symmetric Connection Detection

SNMP - Simple Network Management Protocol

SRAM - Static Random Access Memory

SYN - Synchronize TCP packet flag (start connection)

TCP - Transmission Control Protocol

TDBF - Time-Decaying Bloom Filter

Chapter 1

Introduction

In this thesis we explore the concept of recording previously unattainable network data on edge and core routers. Through an approach based on the new field of network algorithmics we devise a method of reducing the resource requirements of traditional flow tracking strategies, and propose a method of tracking the duration of network flows. Our method of tracking duration can be abstracted to a generalized method of storing state on a per-flow basis. Research into these topics was completed from late 2004 to late 2006 under the supervision of Dr. Chung-Horng Lung at Carleton University in Ottawa, Canada, and as part of a Communications Infrastructure and Technology Ontario internship at Alcatel under the supervision of Peter Rabinovitch.

Our exploration of the world of high-speed routers can begin with a simple analogy. Internet traffic is somewhat like a car. Stand at one intersection for many days and you may see the car pass by several times per day. Stand there for many days or years and there may come a time when the car passes by no longer. After so much time to contemplate the meaning of the car several questions can be answered; How many trips did it make? What was the average number of passengers? When was its first trip, its last? Did it make many trips (like a taxi) or was it low bandwidth (family shopping trips). When tracking network information the car can be equated to a network packet, and the concatenation of all its trips can be equated to a network flow.

Defining a network flow has been approached in several different ways in academic papers, RFCs, and by network equipment vendors. The basic feature common to all definitions

is that a flow identifier is used to define a network flow. Flow identifiers are a combination of packet header fields or other characteristics of network packets. Network traffic is grouped into flows based on the observation that all packets which share a common flow identifier can be classified as belonging to the same flow. Referring back to the car example, one flow identifier is very clear - the license plate number, and other flow identifiers could be the make and/or model of the car. An equivalent standard definition exists for network flows, the standard flow identifier was defined in NetFlow version one as the 5-tuple of IP source and destination, port source and destination, and the protocol number [1]. Other flow identifiers proposed in newer versions of NetFlow and other flow tracking products are based on AS numbers, ingress and egress interface numbers, MPLS labels, and more.

Analysis of the statistics and data generated by tracking and analyzing network flows is one of the primary mechanisms available to ISPs when making networking engineering decisions. ISPs therefore desire to have the most fine-grained data possible on their network traffic. This demand for information from the networking industry has driven an enormous research effort into methods of generating useful network statistics and data. However, in spite of the research accomplished, producing data or tracking state on a per-flow basis remains largely out of our technological reach on high-speed (OC-192 or OC-768) routers. The reason for this lack of per-flow monitoring capability, and the implied complexity of per-flow network monitoring, can be demonstrated by revisiting the car analogy. Instead of a single car, now imagine a busy 10-lane highway, packed with cars traveling over 100kph, so during a typical rush hour 50,000 cars may pass by. That is about 14 cars per second. At this rate, identifying a specific car by its license plate number becomes a complicated task for any person. Likewise, on busy networks the sheer packet rate and number of flows makes recording accurate and fine-grained network data a challenging task.

Ideally we wish to be able to reduce the processing requirements of traffic analysis and data collection without sacrificing accuracy. To this end we describe a filtering technique which is capable of reducing the number of flows, and therefore the computational requirements of analysis applications, by up to 95% for average Internet traffic. Like many other proposed solutions to high-speed network monitoring our solution makes use of a time and

space efficient data structure known as a Bloom filter [2]. The method proposed in this thesis, called Symmetric Connection Detection (SCD), is method of filtering network traffic such that only fully established TCP flows will pass through the filter. SCD uses Bloom filters to maintain minimal state about every TCP connection attempt. The operation of SCD can be summarized as follows; TCP SYN packets are associated to flow identifiers, in a highly compressed format, using two Bloom filters. Once a TCP SYN has been "seen" from both sides of a connection, SCD will report that the connection was successfully established. The unique feature of SCD is its ability to provide very high accuracy while using very small amounts of memory and CPU time. In section 6.2 we show that using only 32k bytes of memory SCD can achieve 99% accuracy even in adverse conditions (900,00 active flows).

Network monitoring applications, such as tracking the duration of TCP flows, can be optimized by using SCD pre-filtering to filter out flows which are never fully established. The reduction in processing requirements is due to the fact that in typical Internet traffic the TCP protocol accounts for 95% of traffic, of which 5-10% is SYN packets ([3] and section 6.1). Reduction in the number of flows can benefit many applications; for example, an application which is tracking the duration of flows requires only those flows which are fully established, and therefore processing any other flows or SYN packets is a waste of computing and memory resources. To further validate this statement we show in section 6.1 that many of the SYN packets seen on the general Internet are not valid connection attempts, but instead are part of DDoS attacks or port scanning. These SYN packets will almost never become established flows. We show in section 6.1 that filtering these incomplete flows can reduce the processing requirements of our hypothetical duration tracking software by 95%.

The second focus of this thesis is tracking the duration of network flows on a per-flow basis. State of the art methods that track the per-flow duration of network flows on high-speed routers are typically about 5% accurate. For example, NetFlow is able to report on the duration of flows, but due to the requirement to sample packets, only high-bandwidth flows are accurately tracked (see sections 2.2.2 and 6.3.2). This inaccuracy puts ISPs at a potential disadvantage when they require flow duration information to calculate

the distribution of network traffic according to the application which generated the traffic, for example, to determine what percentage of their bandwidth is being used by Peer-To-Peer applications. Determining the application-level content of a network flow normally requires Deep Packet Inspection (DPI) of critical packets in the flow. Unfortunately DPI is not available on high-speed routers due to intrinsically high requirements for processing cycles and memory access cycles per packet. An alternative to DPI is to classify traffic based on transport level network flow information, such as flow duration, average packet size, and fan in/fan out [4] [5]. The transport level approach to classifying network traffic has motivated us to develop a method of tracking individual flow duration that is scalable to the processing speeds and resource restrictions present on high-speed routers.

Tracking the duration of network flows allows an ISP to focus on specific traffic types which impact their operational expenses. A simplification to duration tracking can be made by classifying flows into categories according to their approximate duration. This categorization allows an ISP to accomplish their main goal of separating very short flows, short flows, long flows, and very long flows. Equivalently, these duration categories can be thought of as storing some amount of state about each flow in the network, in this case, four states in total. Storing state about individual flows on a high-speed router requires a strategy that is both time and space efficient.

Binned Duration Flow Tracking (BDFT) is a time and space efficient method of tracking the duration of network flows on high-speed routers. Individual flows are placed into "bins", which represent the current state of the flow. BDFT assigns time ranges to each state (e.g. 0-15sec, 15-45sec, 45-75sec, 75-105sec), and moves flows to the next time range (state) on a periodic basis. Assigning the time ranges to each bin is a complicated process which is described in section 5.2.5, for this description we will assume some standard time ranges. A flow which is originally created and assigned the 0-15 second time range is moved to the 15-45 second range after up to 15 seconds and to the 45-75 second time range after 30 seconds. An operator or external program which wishes to know the duration of a flow can query BDFT for the current state of the flow. The state of the flow can then be translated from a state to a duration based on a simple table. BDFT inherits its time and space

efficiency from the use of counting Bloom filters as the data structure which represents the bins. The bin data structure along with several other optimizations allow BDFT to achieve very good performance while using small amounts of memory and computational resources. In section 6.3 we show through experiments using Internet packet traces that BDFT is able to report the correct state for flows 99.84% of the time using only 360,448 bytes of memory, and 99.98% of the time using 720,896 bytes of memory.

1.1 Motivation

The motivation behind our work stems from two factors; first, there is substantial demand from the ISP industry for better information on their networks, and second, the research literature contains little information on tracking the duration of network flows. In this section we explain the motivation from industry, and in subsequent sections we discuss the lack of literature. In section 2.2, we approach the lack of literature by explaining the difficulty of per-flow state tracking and in section 2.4 discuss issues with current network monitoring proposals.

The continual evolution of the Internet creates a dynamic environment for ISPs to operate their network within. Traffic patterns have changed significantly in recent years, from a history of a few heavily used protocols, the Internet today contains hundreds. Identifying and managing the transmission of these protocols is the top priority for many ISPs, as better network management leads to better utilization, which leads to lower costs. Information is the key to network management and design, leading us to focus on an information-poor area for ISPs; flow duration.

Flow duration information can be very valuable both directly to ISPs and indirectly to other network processing programs. Several papers that have studied the average flow duration of P2P and content delivery systems report that average flow durations are over two minutes. The mean duration of eDonkey download connections is 851 seconds [6], the mean duration of kazaa connections is 130 seconds [7], and the mean duration of BitTorrent is over 1000 seconds for successful downloads [8]. Flow duration has also been cited in

numerous papers as a characteristic that can be used to classify flows according to their application type [9]. To the best of our knowledge BDFT is the only acceptably accurate and real-time method to track the duration of flows on high-speed routers.

Network operators require the fastest response to problems or new network traffic possible. For this reason there has always been a desire for real-time reporting of statistics, but most algorithms have relied on non-realtime off-line processing of exported NetFlow records, or other large data stores. BDFT allows queries for the duration of a flow in real-time, without any need for further processing of data.

Many network traffic analysis applications intended for network engineering and other applications by ISPs require computationally intensive processing of flow records. Given that many flows are not complete, since they are part of port scanning or other TCP-based attacks, it is an unnecessary use of resources for flow tracking software to track these incomplete flows. Therefore, flow tracking software (such as BDFT) should not operate on raw network flow data streams, but instead should operate in conjunction with our second development, Symmetric Connection Detection (SCD). By filtering out the up to 95% of connections that are incomplete from typical Internet traffic, a pre-filtering mechanism makes previously infeasible flow tracking applications possible. To our knowledge there are no other incomplete-connection filters that have been published in the literature that are suitable for use on high-speed routers. To make the operation of BDFT and other similar algorithms feasible we were strongly motivated to develop a method of reducing their processing (and memory) requirements.

1.2 Contributions

Our first major contribution is BDFT. Binned Duration Flow Tracking (BDFT) provides real-time flow duration estimates for all flows and packets received on high-speed (OC-192 or OC-768) routers. Flow monitoring and duration tracking on high-speed routers requires strict constraints on the amount of memory and CPU time used, which BDFT meets by scaling in constant memory and time to millions of flows. In addition, to the

best of our knowledge, BDFT is the first algorithm able to provide real-time reporting of flow duration on a per-flow basis on high-speed routers (10+Gbps). BDFT also provides network operators with summary network information, such as the real-time distribution of flow durations within their network. Using Internet packet traces to drive our experimental analysis we show that BDFT is 99% to 100% accurate with memory usage from 11,264 to 360,448 bytes. We plan to publish the BDFT algorithm in an upcomming conference. Finally, in recognition of the innovation that BDFT represents, Alcatel has filed for a patent, with Bradley Whitehead as sole inventor, covering many of the concepts in BDFT, and BDFT as a whole.

Our second major contribution is SCD. Symmetric Connection Detection (SCD) acts as a filter for incomplete TCP connections, and therefore can pass complete TCP connections on to a secondary flow processing application. Like BDFT, SCD is designed to operate in the resource-constrained environment present on high-speed routers. SCD scales in constant time and space with the number of flows present in the network, and provides gradual and predictable degradation of accuracy in overload conditions. SCD is the first algorithm that we know of which provides filtering of incomplete flows on high-speed routers. Using only 32k bytes of memory SCD can achieve 99%+ accuracy even in adverse conditions (900,00 active flows). A paper on SCD has been accepted for publication at the IEEE International Conference on Communications (ICC), to be held in June 2007 in Glasgow, Scotland. Finally, in recognition of the innovation that SCD represents, Alcatel has filed for a patent, with Bradley Whitehead as sole inventor, covering many of the concepts in SCD, and SCD as a whole.

As minor contributions we present a detailed analysis of two publicly available Internet traffic traces, including the flow duration distribution. We also present a modification to Time-Decaying Bloom filters (see section 2.3) which allows TDBF to be used to track the duration of flows.

1.3 Thesis Outline

The research presented in this thesis consists of the development of the flow processing strategies, SCD and BDFT. Over the next seven chapters we develop the background, description, and performance of SCD and BDFT. This thesis is laid out as follows;

Chapter 1 - Introduction In the introduction we describe briefly the goals and operation of BDFT and SCD. We also discuss our motivation for pursuing new network traffic analysis strategies. Our contributions to the field are summarized.

Chapter 2 - Background and Related Work Chapter 2 lays the foundations for the discussions in the rest of the thesis. Traditional flow tracking strategies are discussed, along with the difficulty of tracking network statistics on a per-flow basis. The difficulty of implementing flow tracking on high-speed routers is explained with a description of router hardware architecture. We also introduce the basics of Bloom Filters and Time-Decaying Bloom Filters. Published work that is related to our approach to SCD and BDFT is discussed.

Chapter 3 - SCD This chapter presents the operation of Symmetric Connection Detection, both from a high-level, simple description of the algorithm and a detailed discussion of operation. An extension to SCD called Dual-Filter SCD is also presented.

Chapter 4 - BDFT This chapter presents the operation of BDFT, both from a high-level, simple description of the algorithm and a detailed discussion of operation. Several extensions, optimizations, and implementation suggestions are discussed.

Chapter 5 - Mathematical Analysis and Design In Chapter 5 we mathematically analyze the operation of Bloom filters and counting Bloom filters. These results are then extended to encompass the theoretical performance expectations for SCD and BDFT. We discuss strategies for designing a BDFT array and SCD filter, and present an analysis of the computational requirements of BDFT as compared to d-left hashing [10].

Chapter 6 - Experimental Analysis The performance of SCD and BDFT are analyzed using real-world Internet packet traces to drive our implementations of the algorithms. We present our very positive performance results based on these traces, and also describe the characteristics of the traces including the flow duration distributions.

Chapter 7 - Concluding Remarks Chapter 7 summarizes our findings and experimental results. Directions for future work are discussed.

Chapter 2

Background and Related Work

2.1 Router Architecture

The design of a network monitoring application that is intended for deployment on high-speed routers requires an understanding of the underlying router architecture. A proper application or algorithm design must take into account several important restrictions that are introduced by the architecture layout. This means that while some design considerations are hardware-specific, many routers share the same basic architecture and therefore the same design restrictions. In this section we will describe the generalized router architecture shown in Figure 2.1 and discuss the implications of this architecture on algorithm design.

One of the primary resource restrictions on high-speed routers are memory limitations. The restrictions created by both memory capacity, and the access time of DRAM and SRAM, have been presented in several papers [11] [12]. However there are several other lesser known restrictions which can be presented after a brief description of router hardware.

Router hardware is designed around a type of distributed computing concept; a series of hardware modules communicate with each other through a high-bandwidth internally switched network (backplane). The main hardware modules are CPU cards and line cards, each performing a distinct functionality. Line cards present external connectivity to the router (e.g., Ethernet, OC-48), and have on-board packet processing and switching capabilities. The line cards are typically referred to as the "data path". CPU cards control the overall operation of the router, implement routing protocols, and perform processing that

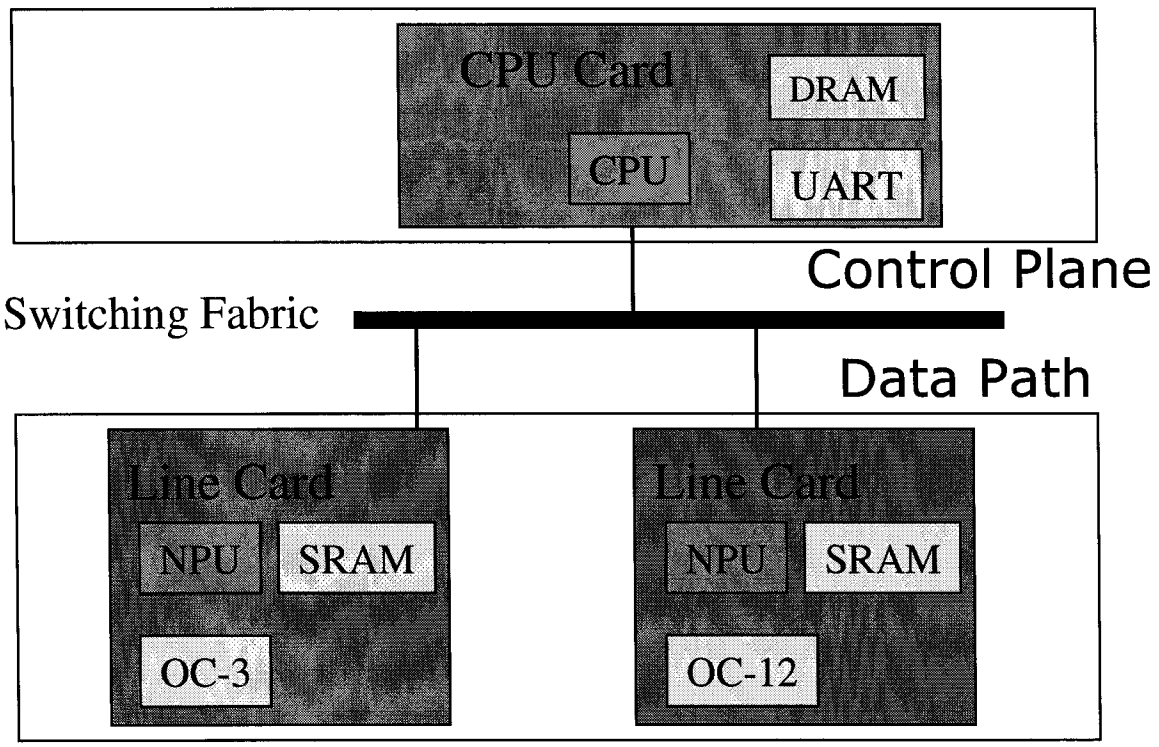

Figure 2.1: Generalized router architecture

affects the router as a whole. CPU cards are typically referred to as the "control plane". Line cards are typically largely SRAM based, and can be augmented with extra hardware such as Content Addressable Memory (CAM). Processing on a line card is accomplished by either custom ASIC hardware or one or more network processing units (NPUs).

The main challenge in implementing a network monitoring algorithm that does not employ sampling is that to acquire direct access to all packet data it must run on a line card. To run on a line card the processing performed by the algorithm must use simple operations only, e.g. bitwise operators (AND, OR, etc), memory read/write, and compares, and minimal branching logic if any. In addition to the computational limitations, memory is limited to only a few accesses per packet depending on line speed and memory speed, meaning that only limited portion of the entire packet may be accessed. Memory space is also severely limited, as SRAM is expensive. An algorithm meeting these processing requirements and memory restrictions can be implemented as an NPU program or in hardware. As a result of

these severe limitations a good compromise design can offload secondary processing to the CPU card, but switching plane bandwidth must be conserved, so only a small percentage of the total line card traffic may be transmitted to the CPU card. A typical amount is 128 bytes for every 10-100 packets, as defined by sFlow [13].

For algorithms implemented on the CPU card the available memory and processing time are less limited. Available memory can be megabytes in size, but the memory is DRAM, so accesses cannot be on a per-packet basis. The CPU used on CPU cards is typically a general processing unit, capable of executing most instructions. Also, typically more CPU time is available than on a line card.

BDFT is a hybrid algorithm which is implemented at both the data path level and control plane, see section 4.3.4. SCD is intended to be implemented at the data path level. Other example algorithms that can operate at the data path level are Space Code Bloom Filters [12], and Bitmap Algorithms for Counting Active Flows [14].

2.2 Flow Monitoring

Flow monitoring as a concept is driven by the need to gather fine-grained information from a network. As discussed in the introduction, the definition of a flow is described by the flow identifier. The flow identifier is typically defined as the 5-tuple of; IP source and destination address, TCP source and destination port, and protocol number. In this section, we discuss the flow monitoring concept in detail explaining why flow monitoring is hard to design for high-speed routers, present currently used strategies for flow monitoring on high-speed routers, and discuss flow monitoring techniques.

2.2.1 Flow Monitoring on High-Speed Routers

The design and implementation of flow monitoring on a high-speed router demonstrates the concept of scalability and its consequences in the real-world. A task that is trivial when a small amount of network traffic is present, and therefore a small number of flows, can become impossible to accomplish when the number of flows grows large. The naive solution

of simply storing all flows in an array quickly becomes impractical when the basic flow maintenance operations are considered. For example, if there are one million flows in an array, then for every packet received by the router a lookup through all one million records must be performed to try and match the packet to a flow. If a few packets are received per second this may be feasible, but on a 5Gbps link with a 625 byte average packet size, one million packets will be received each second, resulting in CPU or memory overload. The scalability problem is also discussed further in several networking and data streams papers (section 2.4.1 and [14]).

Due to the scalability issue, most of the network monitoring implementations which exist today do not support per-flow monitoring for all flows. Instead, the simplest monitoring implements counters which track single statistics such as, the number of bytes received and sent on an interface, or the number of packets. These counters are then exposed to network operators through SNMP. Another approach is Juniper network's implementation of a very coarse-grained flow monitoring counter with per-prefix accounting. In pre-prefix accounting a counter is associated with a specific IP address prefix, and all packets or the number of bytes which match that prefix are counted [15]. These counters are 100% accurate and can therefore be used to drive per-byte billing systems for ISPs.

The Real-Time Flow Monitoring (RTFM) specification describes a fine-grained, flexible, and programmable architecture for monitoring network traffic [16]. This architecture processes every packet, matching it to an existing flow record or creating a new record if the flow is not found. Many different statistics can be calculated based on the resulting records, with no loss of accuracy. However, architectures based on per-flow association of packets, such as RTFM, can not scale to high-speed links such as OC-192 or OC-768 [14]. Currently specialized hardware is required to support lower speed links such as OC-48 [17].

2.2.2 NetFlow and Sampling

NetFlow tracks individual flows by maintaining a table of flow records [1]. Each record must contain flow identification information, and may contain additional information such as the number of bytes for the flow, number of packets, time stamps for the first and last packet,

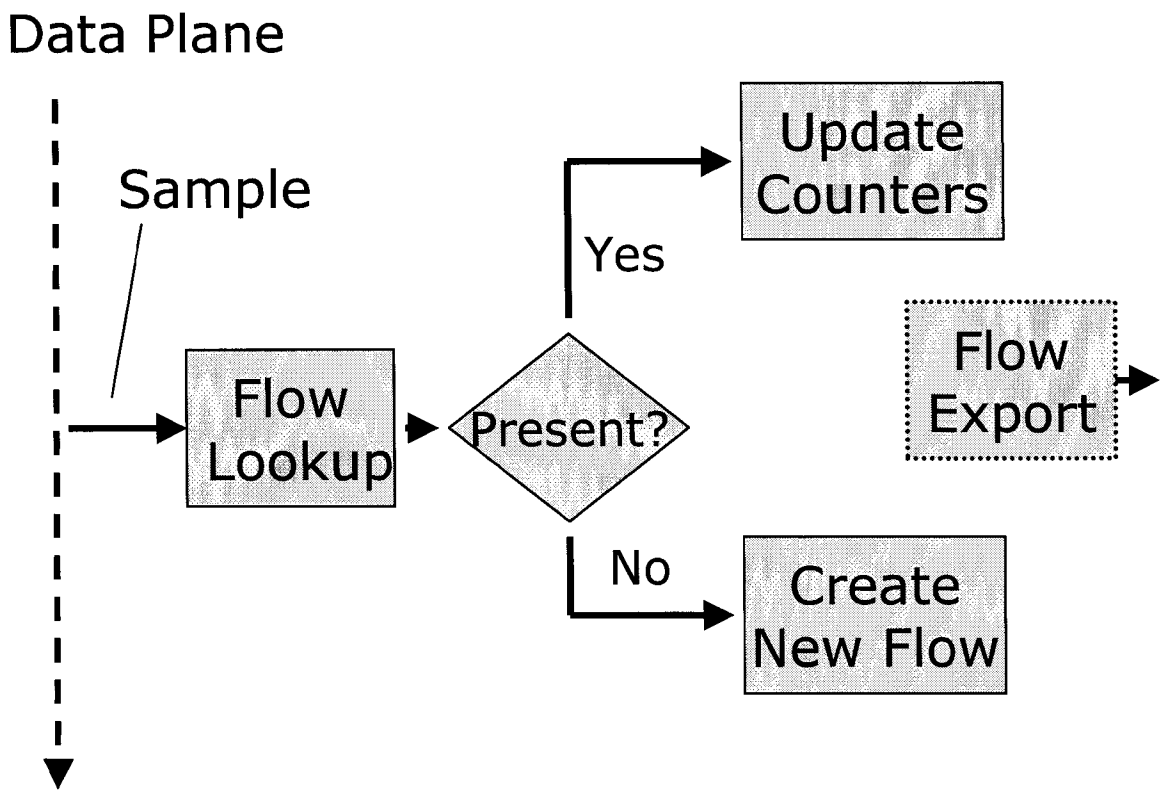

Figure 2.2: Diagram of NetFlow operation

and additional fields as required. Flows are stored in the table until they expire, typically after 15-68 seconds of inactivity or after 30 minutes of continuous activity. Upon expiry the flow record is removed from the table and sent to an external storage server called a collector. Statistics and information are typically only available after offline processing by the collectors, due to the large volume of NetFlow records.

The maintenance operations for the NetFlow flow table are shown in Figure 2.2 To maintain the flow table, packets are read from the network and undergo a classification process to try and associate them with a currently existing flow record. If a flow record does not exist for the flow then a new one is created. If there is no room in the table for the new flow then an older flow is exported to a collector to make room for the new flow. If the flow already exists in the flow table then the counters and other statistics for the flow are updated. Several steps in maintaining the flow tables could be expensive; classifying

the packets, selecting a flow for removal, and the storage space for the tables themselves.

Ensuring scalability to 10+Gbps speeds requires a different approach to the design of the monitoring application. The industry standard flow reporting solution, NetFlow, is implemented on edge and core routers using sampling of packets to reduce processing and memory requirements. Basic sampling selects one out of every n packets for processing, thus introducing substantial inaccuracy in many network statistics [18]. Several techniques have been proposed to increase the accuracy of sampled flow statistics [19], or improve the accuracy for specific applications [20]. These techniques are limited to solving a specific problem, and can only place an upper bound on the sampling induced inaccuracy.

2.2.3 Goals and Issues

An ideal flow monitoring system would have the following characteristics;

- Flexible - Any data set can be recorded for analysis
- Fine-Grained - Information is available on a per-flow basis
- Scalable - Can be implemented on routers with OC-192 and OC-768 interfaces with few resources
- Real-Time - Information and statistics are available through real-time queries
- Low Export Bandwidth - Retrieving information from the router requires little bandwidth

Most existing flow monitoring tools fail on one or more of these points. NetFlow versions one and five have several drawbacks. First, the concept of a flow is fixed, and cannot be modified to suit the requirements of the ISP (this point has been addressed to some extent in NetFlow version 9). A flexible flow definition is perhaps one of the hardest of the ideal points to accomplish. Second, NetFlow does meet the requirement to be fine-grained, but to achieve the scalability required to be implemented on high-speed routers NetFlow requires that sampling be performed. Sampling introduces a high error rate into the NetFlow output

making it unable to accurately report on certain statistics, such as flow duration. The last two points, real-time reporting and low export bandwidth are linked for NetFlow. NetFlow operates by exporting all flow records off the router to an external collection device. This export process is one of the largest drawbacks to NetFlow since the export bandwidth required has been estimated to be 132Mbps for a 10Gbps link. This means that an entire OC-3 connection is required just for the NetFlow output. Processing this volume of data cannot be done in real-time and requires large processing resources on the collection devices in addition to consuming valuable network bandwidth.

2.3 Bloom Filters and Time-Decaying Bloom Filters

A Bloom filter is a bit array which supports set membership tests by using k independent hash functions to address k bits in the bitmap [2]. To insert an item, the k bits the item hashes to are set to 1, and therefore to check if an item is a member of the set, all k bits must be 1 if the item is a member of the set. Bloom filters provide a time and space efficient means of testing if an item is in a set by accepting a small probability of error in the lookup result. Bloom filters can return a false positive, meaning an item is in a set when in fact it is not, but will never return a false negative (an item is not in a set when in fact it is). False positives, false negatives in counting bloom filters, and other properties of bloom filters are further discussed in chapters 3, 4, and error rates are analyzed in chapter 5.

A survey of the network applications of Bloom filters is [21]. Attig and Lockwood have shown that a Bloom filter can be implemented in hardware and can scale to OC-192 (10Gbps) speeds [22]. Attig and Lockwood use a Bloom filter based method to detect patterns in network traffic and report on suspicious flows. A low power strategy is low power Bloom filters [23].

Bloom filters support additions of new items to a set, but do not support removals. To support removals a new type of Bloom filter was proposed called a counting Bloom filter [24]. These Bloom filters have all the characteristics of normal Bloom filters, with the addition of a new type of error; a false negative caused by counter overflow. Counting

Bloom filters are further discussed in section 5.2.4.

Several other types of Bloom filters have been proposed, and the use of Bloom filters for a variety of tasks has increased substantially recently.

Space-Code Bloom Filters (SCBF) provide per-flow accounting of the number of packets in a flow [12]. SCBF is both time and space efficient and is designed to run at the data path level of a high-speed router and use SRAM for storage. Queries can be made of the SCBF of the form "How many packets were received for flow identifier X". In SCBF, the normal k hash functions of a Bloom filter are replicated into M groups of k independent hash functions. When a flow arrives, one of the M hash groups is chosen at random and written to the SCBF. SCBF operation is divided into measurement intervals, at the end of an interval the SCBF is backed up to external storage, and can be queried at a later time. To determine the number of packets in a flow, within a measurement interval, an estimator table is used to relate the number of set hash groups for the flow to the expected packet count. Multi-Resolution SCBFs are also discussed, which are useful for reducing the impact of counter overflow errors. The bins in BDFT are currently implemented using counting Bloom filters, but could be implemented with SCBFs. A major implementation advantage of SCBF over counting Bloom filters is the characteristic of blind streaming, where incrementing the value of a "counter" requires only memory writes and no reads.

Time-Decaying Bloom Filters (TDBF) originate in the data streams field and represent a way to track a distribution of items with regard to time. TDBF are based on counting Bloom filters, incrementing the counters corresponding to an item's hashes each time an instance of an item arrives. The counters are then decremented based on a time-decaying function once per decrement interval. In section 6.3.1 we present a modification to TDBF that allows the duration of individual flows to be tracked. The performance of the modified TDBF is then compared to BDFT performance.

2.4 Published Work

In this section we discuss published literature that is related to SCD and BDFT. For both SCD and BDFT, there is no direct previous work that proposes methods of filtering incomplete flows or tracking the duration of flows. Therefore, instead of directly related previous work we present work that uses concepts similar to those in SCD or BDFT to accomplish other network monitoring tasks. This section is split into three parts, first we present an interesting link between data streams and network monitoring; second we discuss work related to SCD; and third discuss work related to BDFT.

2.4.1 Related Data Streams Publications

Data streams are a generalization of network and database traffic to include common characteristics of all high-bandwidth data transmission paths. The data streams field stems from a paper that relates the space requirements of selection and sorting algorithms to the minimum number of passes required over a data set [25]. In more recent work, S. Muthukrishnan defines a data stream as a stream of information that is received at a very high rate [26], with the following list;

> "High rate means it stresses communication and computing infrastructure, so it may be hard to:
>
> - transmit (T) the entire input to the program,
> - compute (C) sophisticated functions on large pieces of the input at the rate it is presented, and
> - store (S), capture temporarily or archive all of it long term."

This definition of a data stream covers some of the fundamental challenges of most network monitoring applications, presented as scalability issues in section 2.2.3. Network monitoring must ideally; transmit (T): act on all network data; compute (C): record statistics or perform other scanning on the information; and store (S): store the results of all processing at fine granularity. One additional requirement that exists in many network

monitoring applications is real-time reporting of monitoring results. This requirement typically exists in security-critical applications, and adds additional computing requirements to the monitoring application's CPU loading.

The work on data streams potentially offers a way to generalize work on databases, network streams, and other information streams such that some conclusions about one stream type can be applied to other stream types. However, there has been no data streams work related to maintaining flow duration on a per-flow basis. Most of the work in data streams has been focused on database and data mining applications, and as a result there has been no work on network-oriented applications such as tracking network flow duration on a per flow basis. The concept of a flow duration is specific to network traffic, and can not be generalized to, for example, database record inserts and deletes (and vice-versa).

Data streams researchers have published a method of maintaining dynamic quantiles that can be applied to tracking flow duration on a global (not per-flow) basis [27]. A number of data streams algorithms have been developed to track quantiles based on a numeric data base record field. It has been shown that there is an algorithm to find all the quantiles in O(N) time (worst-case) [25]. For the case of an insert-only data stream this has been reduced to O((log eN)/e), where e is the error [27].

Adapting a data streams algorithm to track the duration of current network traffic requires an algorithm which allows both inserts and deletes. Network flow duration information can be modeled as a database containing active flow records with flow identification information and their current duration (e.g. start time and last update time). For each new packet that arrives in the flow, the database record is modified to the current flow duration. When a flow is terminated its database record is deleted. The result is a database that contains the duration of all active flows in the network. Using this model the data stream quantiles or other approaches can be applied to flow duration.

Flow duration quantiles can be calculated for a flow duration database using the RSS (Random Subset Sum) algorithm of A. Gilbert, et al [28]. RSS utilizes a random sampling method to track the quantiles of a large set of data with a small amount of error. For a specific database record attribute (e.g. flow duration) a maximum range is established,

and the range is broken down into evenly spaced segments. A number of random sets are created with each segment having a 50% chance of becoming a member of a set.

2.4.2 Work Related to SCD

After an extensive survey, to the best of our knowledge there is no work directly related to filtering incomplete TCP flows out of network flow data in real-time. This is perhaps due to the relative simplicity of the problem when infinite resources are available to filter traffic. In this section, we discuss other work that has laid the ground work for our extension of Bloom filters and approach.

Stateful packet filters are able to track the connection state of TCP sessions, examples of these are [29] [16] [30]. From the perspective of resource consumption, these stateful filters are equivalent to tracking all flows individually. Storing per-flow state makes these applications very flexible in their feature set, but also requires memory on a per-flow basis. As a result, these applications are unable to process packets at the line speed of a 10Gbps edge router due to, the requirement to use DRAM to store the flow information, and the computational resources required for flow lookup. It will be shown in section 6.2 that even when a stateful filter uses optimization techniques such as a very large hash table to increase flow lookup speed, SCD provides an order of magnitude better performance.

Since SCD focuses on connections that are established, and the opposite problem is detecting connections that are never established, the research into detecting port scanning contains some work that is similar in concept to SCD. However, it should be understood that detecting incomplete connections and reporting complete connections are two different problems. SCD is able to report fully-established connections, but without further processing SCD is not able to report half-open connections. Paxon describes a system called Bro which detects port scans by tracking the number of connection failures for specific hosts [31]. TCP SYN, FIN, and RST packets are used by Bro to track the state of every connection on a per-flow basis (see section 3.1 for a brief description of TCP). Tracking per-flow state requires the use of DRAM to store the large amount of state, so Bro is limited to lower-speed networks.

Weaver, Staniford, and Paxon present a method of containing scanning Internet worms by detecting their port scanning attempts [32]. Again this paper focuses on port scan detection, not established connections. The authors mention using Bloom filters as an approximation cache, but not in the context of tracking connection attempts. Their implementation uses an associative cache to track external connections, and requires a notion of internal and external IP addresses, which would result in inefficient operation on edge or core routers.

2.4.3 Work Related to BDFT

To the best of our knowledge there is no prior work which allows flow duration to be tracked on a per-flow basis, and for all flows, BDFT is the first algorithm to do so. For this reason we have no direct comparison to existing algorithms. In this section we discuss any work that is related to any of the concepts used in the design of BDFT.

Multistage filters, presented by Estan and Varghese, are a method of counting the number of packets in a flow [18] [33]. The naive algorithm to count the number of packets uses an array of counters with a hash function that maps flow identifiers to counters, such that when a packet is received its corresponding counter is incremented. Estan and Varghese extend this concept to the use of multiple arrays of counters with an independent hash function for each array. Each packet received triggers an update of the counters in each array which correspond to the packet's flow identifier. Counters are selected using the hash function associated to each array of counters, with each hash function being independent of the others. When performing an update (increment) to the counters Estan and Varghese propose a conservative update where the counters are updated to "the maximum of their old value and the new value of the smallest counter". This multi-stage update reduces the chances of false positives caused by many different small flows having hash collisions to the same hash value in one array of counters. This concept of using multiple independent hash functions is similar to the operation of counting Bloom filters, which BDFT uses as the basis of its bins.

Estan, Varghese, and Fisk developed an algorithm to count the number of active flows

on a high-speed link [14]. They present a family of bitmap algorithms which use very low amounts of memory, but operate with significant accuracy. They suggest the use of two bitmap data structures; multi-resolution bitmap, and adaptive bitmap. All bitmap algorithms operate on the principal of using a hash function to map a packet's flow identification information to turn on a single bit in the bitmap. The number of active flows can be tracked with a simple counter and a test to see if a flow's bit is already on, in which case the counter is not incremented. Removal of flows can be accommodated by turning the bitmap into an array of counters. Multi-resolution bitmap uses a scheme of dividing the bitmap into regions of update likelihood. High likelihood (low resolution) regions are likely to receive updates, whereas high resolution regions are unlikely to receive updates. To count the total number of active flows the highest resolution region with a significant number of flows can be counted and multiplied by the likelihood of an update to that region. This paper highlights the usefulness of bitmap-type algorithms for measuring traffic information on high-speed routers.

As mentioned previously, NetFlow has significant problems achieving high accuracy for flow statistics due to the requirement to sample. Estan, Keys, Moore, and Varghese have proposed a number of significant changes to the design of NetFlow to improve performance when the router is under attack, and improve collaboration with external network monitoring tools [19]. At the heart of their proposal is the idea to vary the sampling rate according to the current traffic load, resulting in the ability to efficiently use and cap the memory and CPU resources used by NetFlow. Making their algorithm work requires the use of time bins to maintain the accuracy of flow statistics. However, the use of binning means that "the timestamps in flow records can not be used directly to derive flow duration information." BDFT extends the idea of time bins to group flows according to their duration. By reporting estimated flow durations BDFT is complimentary to the proposals in "Building A Better NetFlow."

Counting the number of active flows from sampled flow statistics has been shown to be infeasible. Duffield, Lund, and Thorup devised a clever work around to this problem by including the use of protocol-specific information in their algorithms to estimate flow

distributions [20]. For TCP flows the NetFlow records are checked to see if the SYN flag (or other indication of the start of a flow) was present in the packets sampled for the flow. This information is used to estimate the total number of flows based on the sampling rate and number of sampled flows that contain a SYN. BDFT follows this example by making use of protocol-specific information to increase the accuracy of its flow duration estimates. BDFT makes use of further TCP-specific flags; FIN, and RST.

In SIGCOMM 2006 Bonomi, et al. [34] published a method of tracking the state of network flows which is similar in function to BDFT abstracted to track state instead of duration [34]. "Beyond Bloom Filters" describes the use of Bloom filters and d-left hashing to enable per-flow tracking of state in a network. They present two variations of a state tracking system which use Bloom filters. Their first method, called Direct Bloom Filter (DBF), uses a single counting Bloom filter to store a set of <flow, state> pairs. In DBF, to lookup the state of a flow, the current state of the flow is required or all possible states must be searched. The probability of false positives in DBF is expected to be high due to the requirement to perform many searches on a single heavily loaded filter. If the state of the flow is not known for searches and removals, as is the case when tracking flow duration, the DBF is not suitable. Their next approach uses counting Bloom filters to store both a count and a state in each cell corresponding to a flow's hashes, called Stateful Bloom Filter (SBF). If a cell has a count greater than one, then a collision has occurred and that cell can be ignored. State lookups will succeed if at least one of the cells has a count of exactly one. For SBF the probability that a cell will have a multiplicity of two is quite high, resulting in the SBF returning "dont know" for the state of many flows. Their last approach is based on d-left hashing [10] and fingerprints. Each cell in the d-left hash stores the fingerprint and the state of the flow. There are also proposals for eliminating the probability of a fingerprint collision between the cells. The d-left hashing approach has the potential to be very accurate up to about 80% memory efficiency, but will require higher computational resources than BDFT. The computational efficiency of BDFT, and the d-left hashing based approach, are analyzed in section 5.5. Beyond Bloom Filters was published late in the development of this thesis, after the research phase was complete. For this reason the error-rate performance

of BDFT is not compared to the "Beyond Bloom Filters" proposals through experimental analysis, and is left for future work. We expect that due to the generalized nature of these proposals a specialized method of tracking flow duration such as BDFT will offer higher performance.

Chapter 3

Symmetric Connection Detection

Symmetric Connection Detection (SCD) provides a 95% reduction in the number of flows which must be tracked and processed by a per-flow network monitoring algorithm. This reduction is accomplished by reporting when a TCP or other connection-oriented connection attempt is very likely to result in a fully established connection. When used as a filter, SCD is able to filter flows which are never fully established, and therefore pass only those flows which are fully established to a secondary processing algorithm. In section 3.1 we give a high-level overview of the operation of basic SCD, and in section 3.2 we describe an extension to SCD to improve accuracy. Chapter 5 provides a detailed analysis of SCD performance and design criteria (e.g., Dual-Filter SCD time ranges) which complements the high-level overview in this chapter.

3.1 Connection Detection Overview

Two fundamental requirements can be identified for any algorithm that implements a filter which passes only complete connections. First, every network packet must be processed and some amount of state must be stored for every connection establishment attempt. Storage of the connection state is necessary for the algorithm to track the connection progress of the endpoints, and determine when a connection is either fully established or is very likely to become fully established. Second, a detection mechanism must decide when the flow establishment process is complete by monitoring or comparing the state of all flows.

Based on these two fundamental requirements the basic operation of SCD is straightforward to describe. SCD stores the state of all connection attempts and performs a comparison on the connection state to determine when a connection has been established. SCD can report the current connection status in real-time, every time the state of a flow changes. The connection status is reported as a boolean value; true if the flow is now established, and false if it is not yet established. Connection information can then be used to filter or pass packets for that flow to a higher level monitoring system, or the statistics can simply be logged and provided to network operators.

The process of tracking the connection state may be specific to the underlying protocol being tracked. In this section, we assume that the underlying protocol is TCP, and therefore begin with a short description of the TCP connection process and how it relates to SCD. To establish a TCP connection a three-way handshake process takes place; each computer sends a SYN, and the initiating computer sends a SYN-ACK to complete the connection [35]. Once the SYN-ACK is received the connection process is finished and the TCP session is fully established. Tracking the establishment of a TCP connection therefore requires keeping track of all three states, however this can be simplified to two states with the following observation. From a point in the middle of the route between the connection endpoints the receipt of SYN packets from both sides of a connection implies that both computers can reach each other and want to establish a connection, strongly indicating that the connection will be established with a completing SYN-ACK. SCD makes use of this observation and defines an established connection as one where both sides have received a SYN from the other side but not necessarily a SYN-ACK. Therefore SCD processes only TCP SYN packets, or an average of about 1 in 20 packets (TCP SYN packets are about 5% of network traffic as discussed in section 6.1).

The problem of tracking connection establishment can now be defined as the following question; when a TCP SYN is received from one side of a connection has the other side already sent a SYN? If so then the connection is established, if not then store the fact that this side of the connection has sent a SYN. To answer this question SCD keeps state on all SYN packets that have been sent and the direction that they were sent in. Direction

is determined by comparing the source and destination IP addresses, e.g. if source IP is greater than destination IP then the packet is assigned direction 1, and if source IP is less than destination IP the packet is assigned direction 2. Storing the flows which have sent a SYN in a specific direction could be accomplished through the use of many different data structures, but many potential data structures would lack sufficient performance to be able to keep up with the requirement to perform a search and possibly an insert on every SYN packet. Therefore, the data structure must be time and space efficient, and ideally would support searching and inserts that scale in constant time with the number of items stored. Bloom filters are such a data structure.

SCD is designed to operate in a resource-limited environment, and undergo gradual degradation of accuracy as resources become more limited. This operation is accomplished through the use of Bloom filters. The only data storage required by SCD is the SYN-direction information for each flow. To meet this storage requirement we employ two Bloom filters, one filter for each SYN direction. Bloom filters represent a set that can be tested for membership. Mapping this concept to our problem can be done as follows; when a SYN is received, test the Bloom filter for the opposite direction to see if a SYN was sent from the other side; if so, the connection is established. If a SYN has not yet been received from the other side, then the connection is not yet established, and this is either the first SYN packet in the connection or the other side is not responding. If the flow was not already stored in the filter for its direction it is added. See section 5.3 for a detailed analysis of the theoretical performance of SCD and filter sizing guidelines.

Figure 3.1 is a flow chart describing the processing of a TCP SYN packet by SCD, with the following steps:

- **1. Compute Src > Dst:** The source and destination IP addresses are compared as unsigned integers to determine which address is greater, source or destination. If they are equal the packet is assumed to be corrupt, and is ignored.
- **2. Lookup IP:** The Bloom filters are queried to see if a SYN packet was sent in the opposite direction for this flow, e.g., if the incoming packets source IP is greater

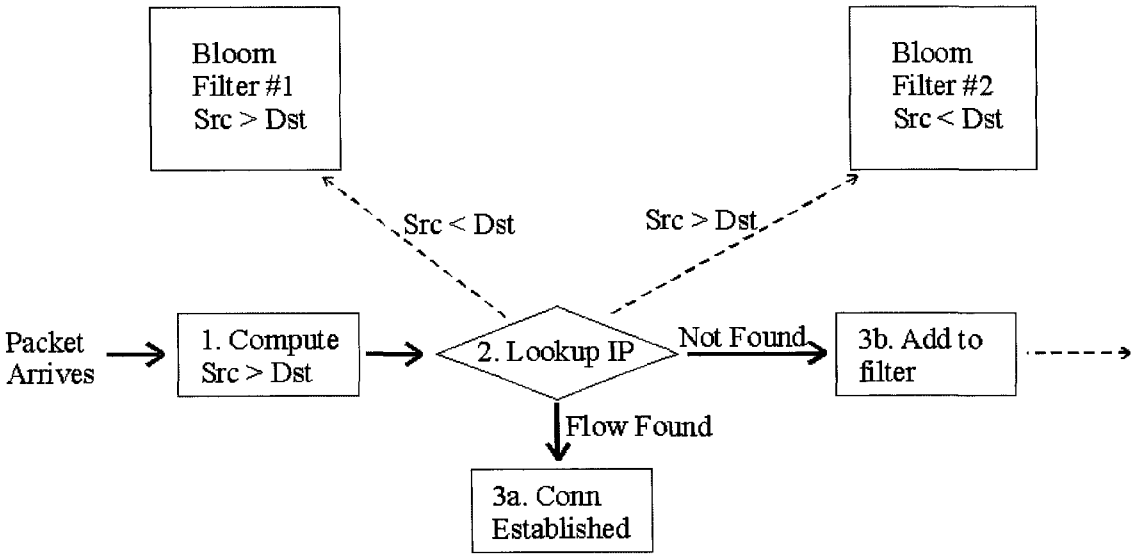

Figure 3.1: Flow-chart of SCD operation

than the destination IP, then the Bloom filter for the opposite direction (Dst > Src) is queried. The packet is used to generate a number of hashes which are used to query the Bloom filters. The number of hashes used is a parameter.

- **3a. Connection Established:** If the Bloom filter returns a positive result to the query, then it can be concluded that the connection is established, to a high degree of accuracy.

- **3b. Add to filter:** If the Bloom filter returns a negative result the corresponding Bloom filter is updated with the flow, e.g., if the incoming packets source IP is greater than the destination IP, then the flow is added to the Src > Dst Bloom filter.

A potential problem arises when continual operation of SCD is considered. If left unchecked, the Bloom filters representing each direction would eventually become full and false positive error rates would climb to unacceptable levels. To avoid this situation the Bloom filters can be cleared of all their data on a periodic basis. The length of this period is the third parameter of SCD, the maximum connection time. The maximum connection

time describes the maximum time that a TCP connection can take before it is no longer tracked by SCD. If the connection establishment exceeds this time the connection becomes a false negative due to the filters being cleared. A false negative occurs if the connection state is lost when the filters are cleared, because the recorded state verifying that the original SYN was sent is erased, resulting in SCD reporting that the flow was never established (a false negative). This raises a potential issue; the minimum connection time that will report a false negative is potentially 0 if the original SYN packet was received by SCD just before the filters were cleared. We call the minimum time that a TCP connection can take to complete before being lost when the filters are cleared the lower bound of the maximum connection time. This leads us to propose an improvement over basic SCD, dual-filter SCD.

3.2 Dual-Filter SCD

The connection establishment phase of TCP can range from a few milliseconds to several minutes. This extreme variability in the connection establishment time for TCP is one of the major sources of error in SCD. Connections which take much longer than normal to complete (more than a few seconds) can become false negatives if they exceed the lower bound of the maximum connection time. Dual-Filter SCD reduces the number of errors caused by this variability by raising the lower bound of the maximum connection time from zero to half of the upper bound of the maximum connection time.

Dual-filter SCD modifies basic SCD from one Bloom filter per direction to two Bloom filters per direction. Each Bloom filter contains the state of connection attempts for a non-overlapping portion of the total range of time. For example, if the SCD maximum time is 10 seconds one filter would initially cover the 0-5sec range and the other would track 5-10sec. Flows are moved between filters by an aging process, which will be described below. After 10 seconds of running time the newer filter will contain flows for the past 0-5 seconds, and the older filter would contain flows from 5-10 seconds. During operation of SCD new SYN packets are recorded in the newer filter, and the older filter simply maintains the state of older connection attempts. Upon receipt of a new SYN packet, queries for membership to

check if the connection is now established are performed against both the older and newer filters.

As with standard SCD, Dual-Filter requires an aging process to prevent the build up of out of date flow data and maintain accuracy of the filter. Each filer has a lifetime which is half of the maximum connection time. Aging occurs when a filter has reached the end of its lifetime, which is five seconds in our example above. The aging process moves the newer filter to the older position (which can be as simple as updating a pointer), and clears the older filter and moves it to the newer position.

The aging process is as follows, and is repeated once for each direction;

1. The older filter is cleared. Any flow information that was in this filter is lost.
2. The newer filter is aged to become the older filter, possibly by simply updating a pointer
3. The older filter is recycled to become the newer filter.

As a result of the dual-filter setup, flows that are received just before the aging process takes place will be moved to the older filter. Once in the older filter the connection state will be maintained until the next aging occurs, therefore the lower bound of the maximum connection time is increased to half the maximum connection time (5 seconds in our example).

Chapter 4

Binned Duration Flow Tracking

Binned Duration Flow Tracking (BDFT) is a method of tracking the duration of network flows on a per-flow basis, for every flow that passes through a network device. In this section we present a high-level overview of the operation of BDFT, followed by a detailed description of permissible BDFT operations. Extensions to BDFT which can improve accuracy and computational performance are also presented. Chapter 5 provides a detailed analysis of BDFT performance and design criteria (e.g., bin time ranges) which complements the high-level overview in this chapter.

4.1 Flow Duration Tracking

The operation of many network measurement applications can be abstracted to a requirement to store some amount of state about individual flows. The storage of state on a per-flow basis presents a challenge on high speed routers due to the requirement to store both the state itself and the flow identification information related to that state. In the case of NetFlow-style flow records this per-flow state is a minimum of 21 bytes for the start and end timestamps (the state), and the flow-identification information (standard 5-tuple of IP source and destination, port source and destination, and protocol type). Given that standard Internet-mix traffic has about 200,000 active flows per Gbps, this results in peak memory usage of 42MB on a 10Gbps router. This memory requirement is beyond of the

current capacities of SRAM, making the NetFlow based approach infeasible based on memory usage alone, with the processing requirements only adding to the infeasibility. BDFT provides a clever workaround to the problem of storing per-flow state.

Duration-BDFT is a data structure and algorithm designed to track the approximate duration of all TCP flows seen on a high-speed router. BDFT can also be extended to any network measurement application where minimal state is required, and where the operations required are adding flows, removing flows, and querying for a flow's state. Compared to NetFlow, BDFT reduces the memory requirements in two novel ways. First, the flow identification information is simply not stored in memory. This approach to network monitoring, not storing the flow identification information, is one of the primary contributions of BDFT, and is explained further below. Second, the per-flow state information is stored by splitting the flows into a number of bins, each bin is associated with some state, so all flows in a bin share some common characteristic. Bins are the only data storage component of BDFT. In duration-BDFT each bin represents an independent and arbitrary length of time, therefore the bin that a flow is in corresponds to its current duration. A bin can be represented and stored using an arbitrary data structure, however selection of an appropriate data structure is a critical design decision. The selected data structure must allow the flow's state to be saved, queried, and removed, without any requirement to store flow identification information.

In duration-BDFT (from this point all references to BDFT are of the duration variant), counting Bloom filters were selected as the default bin data structure. The selection of a Bloom filter variant as the bin data structure allows the intrinsic operation of Bloom filters to be used to our advantage, by replacing the flow identification information with hashes. Counting Bloom filters operate by selecting a number of independent hash functions which are used to index counters in an array, incrementing them on insert, and decrementing them on delete. When a packet is received the hashes are calculated based on various flow identification information contained in the packet header. The hashes are the only information required to index the flow's location in the filter. Therefore, all flow identification information is calculated on-the-fly from the current packet and the hash values, not stored

in memory. In our reference implementation of BDFT, counting Bloom filters were used; however, it is possible that higher performance could be achieved from the use of other data structures such as space code Bloom filters.

BDFT's bins are its only data storage component, so the execution of BDFT involves operations that modify, or lookup, the data stored within the bins in response to external inputs. These operations are triggered by TCP packets received in the data path of a router (or other networking device) with the SYN, FIN, or RST flags set. The search operation is normally triggered by an external operator or software agent. When a TCP SYN packet is received, the corresponding flow is entered into the first bin, and is then automatically aged to successively older bins until the flow is removed when a FIN or RST packet is received. To determine the current or end-point duration of a flow, BDFT determines the bin number that the flow is in, which is then translated into a range of time (or the midpoint of the bin) and returned to the requester. The next sections describe the operations of BDFT in detail.

4.2 BDFT Operations

In this section we describe the operations that define how BDFT maintains duration information for all flows in a network. We also describe how BDFT can be queried to determine a flow's duration. Figure 4.1 shows the operations described below (Figure 4.2 and Figure 4.3 also show aspects of these operations and may be helpful references);

Add a Flow Flows are added to Bin #1 when they enter a "partially established" state, which we define a receipt of a SYN packet from either side of a connection (1st or 2nd step of the TCP 3-step handshake). Flows are added by creating k hashes from the flow identification information, searching all bins to see if the flow already exists, and if not incrementing the counters in Bin #1 corresponding to those hashes. Searching all bins can be avoided by using SCD pre-filtering as described in section 4.3.1, and flows are only added on the 2nd step of the handshake when filtered with SCD. When adding a flow to a counting Bloom filter, it is possible that one or more counters are

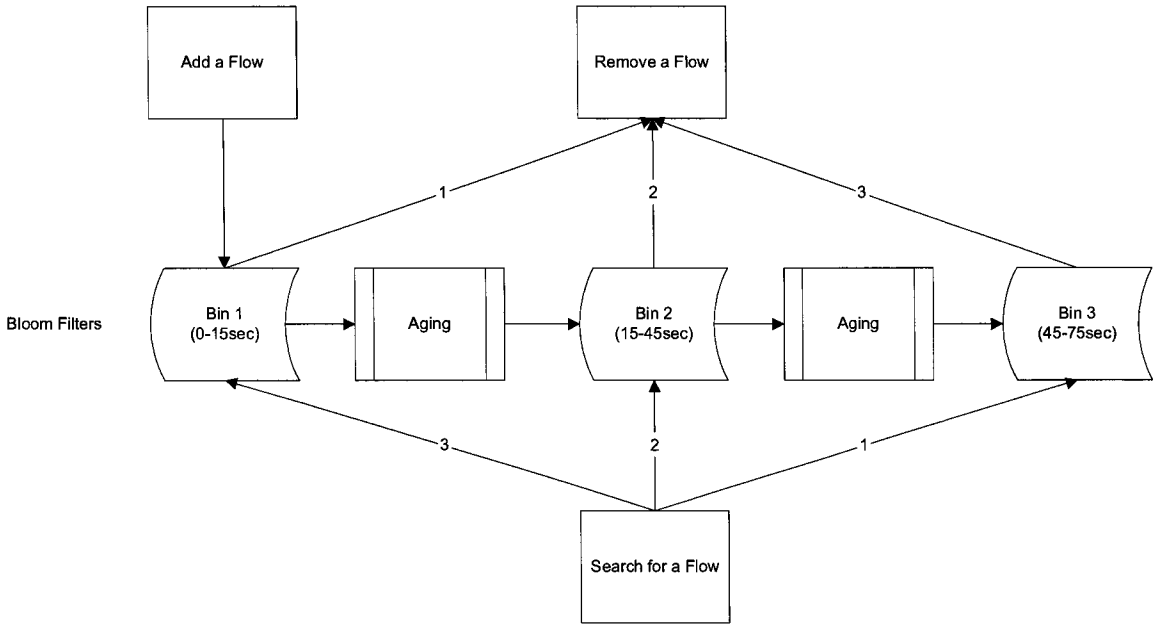

Figure 4.1: Diagram of BDFT operations

already at their maximum value. In this case counters which are at the maximum value should be left at the maximum, and all other counters incremented. Ideally each flow should only be added once, and flows which are never established should not be added (e.g. port scans). Flows which never complete the establishment phase must be removed from the filter.

Remove a Flow TCP packets containing a FIN or RST flag signal the end of a flow, at which point the flow is removed from its bin. Flows are removed by searching from the shortest-duration bin to the longest. When the flow is found the counters corresponding the flow's hashes are decremented. The counters corresponding to the flow must be decremented every time a FIN or RST is received, until one of the counters reaches zero. This results in an aggressive removal of flows; e.g. some flows may be removed prematurely, due to multiple FIN packets being sent. A solution for multiple removal of flows is presented in section 4.3.1. Bins are searched starting with the youngest based on the observation that 50% of flows last less than 2 seconds,

and 90% last less than 45 seconds [36], so the flow will most likely to be found in the youngest bin. An alternate removal strategy which reduces errors due to false positives is presented in section 4.3.3. When a flow ends, a notification can be sent to an external agent.

Aging BDFT maintains its per-bin state by "aging" - the process of moving all flows in a shorter-duration bin to the next longer duration bin. When a bin is in a state where it needs to be aged we say that it is "expired". Each bin represents a time range (duration) for flows. The time range which each bin represents must be selected based on the accuracy required vs. memory requirements and is discussed further in section 5.2.5. As time advances during the operation of BDFT, the flows in a bin become older, until the oldest flow in the bin is older than the time range of the bin, at which point the bin must be aged. The aging process is key to BDFT, it allows the maintenance of the state for all flows. By keeping flows in counting Bloom filters, and aging the filters in time, no flow-specific information such as flow start time needs to be kept. When a bin is expired the flows that are currently in the bin are moved to the next longer duration bin, as shown in Figure 4.2. Depending on the implementation of the bins this may mean individually copying counters to the next bin, or simply updating a pointer to each bin.

Search for a Flow Searches are performed by an external agent submitting flow identification information which can be used to generate the k hashes. Searches are performed starting with the oldest bin first, and moving to sequentially younger bins, until a bin is found where all counters are greater than zero corresponding to flow's hashes, at which point the flow is "found" (or a false positive was found). The duration for the flow is an estimate calculated by determining the midpoint time for the bin the flow was found in. For example if the flow is in a bin with a time range of 45-75 sec bin a duration of 60 sec would be returned.

Figure 4.3 shows an example of the life of a flow as it is inserted, aged, and removed from BDFT. In this example BDFT has bins with example time ranges of 0-15sec, 15-45sec,

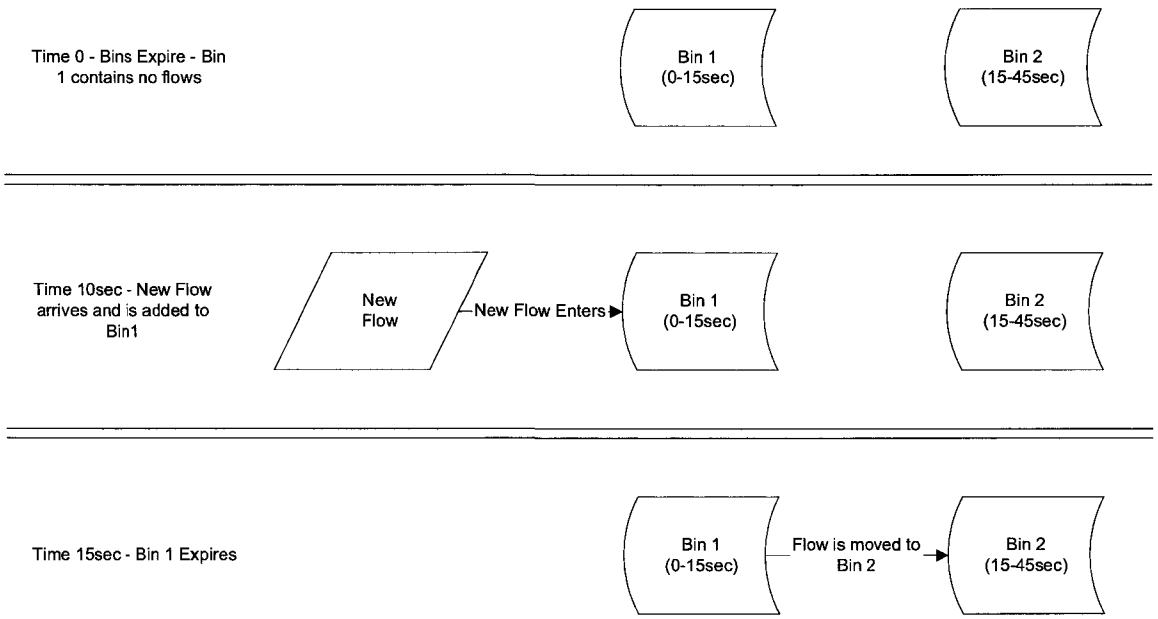

Figure 4.2: BDFT aging process

45-75sec, and 75-135sec, and the flow lasts for 55 seconds. The flow arrives just after Bin 1 was aged, and therefore the flow will be in Bin 1 for its full 15 second duration. In normal operation the timer to expire Bin 1 is always running, and therefore, for example, a flow could arrive when Bin 1 is just about to expire or at any other time. In Figure 4.3 the "Expire Bins" box denotes the continuous process of checking all bins to see if they need to be expired.

The following steps describe the BDFT operations taking place;

1. The new flow arrives
2. Its hashes are calculated based on IP Src/Dst, Port Src/Dst, and protocol type
3. The flow is added to Bin 1 by incrementing the counters corresponding to the hashes
4. After 15 seconds Bin 1 expires and its flows are moved to Bin 2
5. After an additional 30 seconds Bin 2 expires and its flows are moved to Bin 3

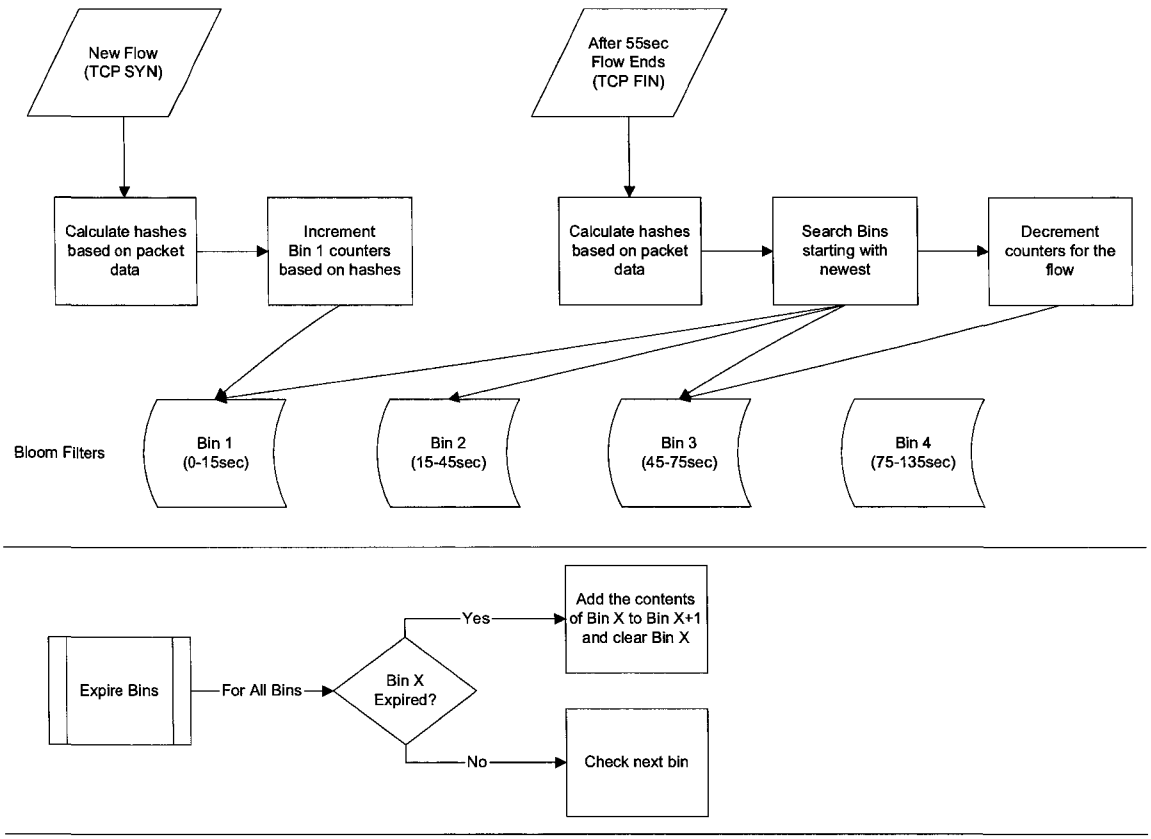

Figure 4.3: Flow chart of the life of a flow in BDFT

6. After 55 seconds from the flow start, a TCP FIN is received for the flow, and the removal process begins
7. The flow's hashes are calculated as above
8. The Bins are searched for the flow's hashes starting with Bin 1
9. The flow is found in Bin 3, so the counters corresponding to the hashes are decremented in Bin 3

4.3 BDFT Extensions

The standard BDFT as described in section 4.2 will provide a basic level of performance suitable for some applications. For applications requiring higher performance in terms of

accuracy there are numerous ways to increase the accuracy of BDFT substantially. This section presents several enhancements to both the accuracy and computational requirements of BDFT. Many of these enhancements come at very little cost in terms of implementation complexity or computing requirements, and therefore should be employed in most BDFT implementations.

4.3.1 Enhanced Insertion and Removal

The accuracy and computational performance of BDFT can be improved if one can guaranteed that for each flow received, the insert and remove functions will be executed once and only once. To be able to guarantee only one insert and removal per flow, the SYN/FIN/RST packets cannot be used *directly* to control algorithm operation, as is done in standard BDFT. SYN, FIN or RST packets are not reliable due to timeouts and lost packets, so a mid-point router may not observe the same packets as the connection end points. Having an unknown number of SYN/FIN/RST packets can lead to a case where there is an imbalance between the number of SYN and FIN/RST packets. Since BDFT increments counters on SYN packets, and decrements them on FIN/RST, this imbalance can cause incomplete removal of flows (SYN > FIN/RST) or potential removal of multiple flows which share hashes by setting one of the shared counters to zero (FIN/RST > SYN). Another effect of multiple SYN packets per flow will be discussed in section 5.2.4; BDFT's accuracy is reduced if any counters overflow, so in the ideal case the counters would only be incremented once per flow, and not once per SYN packet.

Balancing the number of SYN packets vs. the number of FIN/RST can be accomplished using pre-filtering mechanisms such as Symmetric Connection Detection (SCD). Using SCD also has the additional benefit of ensuring that BDFT counters are only incremented once per flow. Section 3.1 describes the basic operation of SCD, which includes the ability to monitor all active flows in a network and report only those that are fully complete to a higher-level monitoring program, such as BDFT. Therefore SCD can be employed to meet the guarantee of only one insert per flow, but balancing SYN vs. FIN/RST also requires that only one removal notification be sent to BDFT per flow, regardless of the number of

actual FIN/RST packets. The functionality required for FIN/RST can also be achieved using SCD or a simple variant. A pair of Bloom filters could be employed to track all of the flows which have already sent a FIN or RST, by adding the flow to the filter and notifying BDFT of the removal if the flow is not already in the filter. These FIN/RST Bloom filters could be maintained using a time-based rotational scheme similar to Dual-Filter SCD (see section 3.2). BDFT's accuracy can be increased substantially using these techniques (accuracy increases are discussed further in chapter 5), therefore the the small incremental memory cost to implement pre-filtering such as SCD in conjunction with BDFT is a good design tradeoff.

4.3.2 TCP Timeouts - no FIN or RST

BDFT relies on the receipt of FIN or RST packets to signal the end of a TCP connection and remove the flow from its present bin. However, it is possible for a TCP timeout to occur such that no FIN or RST packet is ever sent on the connection. For example, when an Internet connection goes down or a route changes, so packets no longer reach the router running BDFT. In this case the flow is "hung" in BDFT and will never be removed. Eventually hung flows overwhelm the long duration bins resulting in a dramatic loss in accuracy for both long duration and short duration flows. As a result, timeouts must be accounted for in environments where they are possible, typically about 0.1% of Internet flows result in a timeout (see Table 6.3).

A simple approach to handling timeouts is to increase the size of the longer duration bins to account for the extra flows, and "flush" the longest duration bin periodically. For example, in a case where flows up to one hour in duration must be tracked, the oldest bin in BDFT may have a lifetime of ten minutes, so the oldest bin would cover the 50-60 minute range. When the oldest bin expires, its flows are no longer tracked, so flows which have undergone a TCP timeout are flushed out of the BDFT array when the bin expires. This solution should be acceptable for most implementations when a small increase in memory usage is not a problem.

An alternate approach is to track all flows and detect when they undergo a TCP timeout

(normally defined as two minutes of failure to respond to a request [35]). This tracking can be accomplished through the use of a Bloom filter in combination with high-rate sampling (1 in 10, or higher) of all packets. For every sampled packet the flow is entered into the timeout-tracking Bloom filter, so the filter contains all the flows which have sent packets in the last interval. The interval should be equal to the lifetime of the longest duration BDFT filter. At the end of the interval the timeout-tracking Bloom filter is cleared. When the BDFT aging process takes place the hashes which are being aged should be checked against the timeout-tracking filter. If the hashes are not present (e.g., a packet for the flow has not been received in the last interval) then those hashes which are not present in the timeout-tracking filter and are present in the bin being aged should be removed from the bin that is being aged. This solution provides early removal of flows which have timed out at the expense of processing a much higher percentage of network traffic.

An additional source of timeouts is a change in routing path. BDFT requires that all packets related to a connection pass through the network device that BDFT is running on, and that the routing paths do not change.

4.3.3 False Negatives and Effects of False Positives

In some situations it is possible for BDFT to produce a false negative. This is a serious error which is caused be several factors which will be explained in this section. A basic characteristic of Bloom filters is that they can produce false positives but will never produce a false negative result. The counting Bloom filters used in BDFT can produce a false negative, but only if a counter overflows, so care should be taken to ensure no counters overflow (also see section 5.2.4). These failure modes apply when a Bloom filter is used independently, however when Bloom filters depend on each other, as in BDFT, further error modes can arise.

The most prevalent error mode which can result in a false negative in BDFT is a removal attempt that removes a flow from the wrong bin. This occurs when a removal search generates a false positive, for example, that a flow is in bin three when it is actually in bin ten. In this case the flow will be removed from bin three in error. The counters that

will be decremented in bin three when the flow is removed actually belong to other flows, which will have one or more of their counters decremented. The affected secondary flows are now false negatives as they have been partially removed from their bin. In addition to the flows being false negatives, they can now never be removed from the bin, so the counters corresponding to their other hashes will never be decremented.

Reducing the affect of the false negative problem can be done in several ways. We call these methods false positive correction, since they reduce the affect of false positive removals, and when used in conjunction with BDFT we call them BDFT-FPC. A simple approach to false positive correction is to try and reduce the number of false positives encountered in removal searches by searching the most accurate bins first. Normally this means searching the longer duration bins before the short duration bins. This is not an ideal solution since most flows will be added and removed from the shortest duration bin, and therefore searching longer duration bins is a waste of computational resources. This solution can still result in false positive removals of longer duration flows.

Another approach is to maintain a list of any removals that are considered to potentially be false positives. This approach relies on the fact that a false negative can never occur during normal operation of a Bloom filter, so when a false negative does occur it is known that there was a false positive removal associated with this flow. When this situation occurs the list of potential false positive removals can be checked to determine which removal caused the false negative, based on the counters that were decremented and the bin that they were decremented in. The false removal operation can then be reversed and the correct removal performed.

A third solution to this problem is the use of backup Bloom filters to track those entries which are known to be false positive. The idea of using backup Bloom filters is derived from work presented in the "Bloomier Filters" paper [37]. A small Bloom filter is maintained representing all flows in the BDFT array, this filter must be equal in size to the smallest Bloom filter in the BDFT array. Before a flow is inserted into the BDFT array the all-flow filter is checked to see if there is a potential conflict between this flow and another flow or set of flows which are already in the array. If there is a conflict, further processing can

be performed to mark the flow as being a potential false positive, for example, by using another Bloom filter. Additional state should be stored about potential false positive flows to identify their hashes and bin. When a removal request is submitted for a false positive flow the additional information about it can be used to determine the actual bin that the flow is in.

4.3.4 Datapath Aggregation

For optimal performance, BDFT must process every SYN, FIN, and RST packet received at the data path of a router or other network device. Given that a SYN/FIN/RST packet occurs once out of every 20 packets in normal Internet traffic, processing every SYN/FIN/RST packet results in a sampling rate of 1:20, which is exceeds the traditional sampling rates of 1:100 or 1:1000 for NetFlow/cFlowd. This raises a potential issue with the BDFT architecture, as 1 in 20 sampling rate could exceed the memory and backplane resources of modern routers. However, the effects of BDFT's required sampling rate can be virtually eliminated by two observations:

- BDFT requires a sample of 3 hashes (12 bytes) per SYN/FIN/RST packet. These hashes are independent of each other, and are all of the information required by BDFT, the actual packet data is not required.

- BDFT samples can be aggregated to reduce backplane and processing overhead, to reduce the number of router backplane transactions.

For example, a small 120-byte SRAM buffer could be used at the datapath level to buffer BDFT samples. A 120-byte buffer would hold 10 samples, and is approximately the same size as the sFlow sampling size of 128 bytes [13]. In effect, buffering samples in this way would reduce the backplane overhead used by BDFT by a factor of 10, resulting in an equivalent sampling rate of 1:200. Larger buffers could be used to further reduce the rate. A 1200 byte buffer would reduce the sampling rate to 1:2000. Also note that hashes do not need to be 32 bits, they could be reduced to 24 bits for further reductions in backplane bandwidth.

4.4 BDFT Variations

The performance of BDFT as presented in this thesis is tied to the performance characteristics of counting Bloom filters, as they were chosen as the data structure to implement the bins with. Other variations of BDFT could use different data structures or combinations of data structures to implement the bins, resulting in increased accuracy (up to 100%), or reduced computational requirements. For example the bins could be implemented as array-backed hash tables for 100% accurate tracking. Another configuration could use d-left hash tables for the bins, or use a counting Bloom filter for the first bin for the reduced computational requirements and d-left hashing for subsequent bins. Section 5.5 presents a comparison of computational performance of bins implemented using d-left vs. counting Bloom filters.

Chapter 5

Mathematical Analysis of Performance and Design

This chapter extends the high-level descriptions of SCD and BDFT given in Chapters 3 and 4 to a low-level mathematical analysis of accuracy and computational performance. The design of a BDFT array is described in detail including exact descriptions of parameters and formulas governing accuracy, and heuristics for the selection of bin sizes and time ranges.

5.1 Analysis Overview

The design of a BDFT array is determined by over five parameters of the algorithm as well as the selection process for bin time ranges. In this section we describe the performance of BDFT through statistical analysis of the effect of algorithm parameters. The performance of BDFT can be characterized in several different ways according to the needs of the end user, but we adopt the metric of expected search error as being representative of BDFT's performance to the largest number of users. The equations and heuristics in this section describe the various trade-offs involved in designing a BDFT array, all of which affect the search error rate. The major sections in this chapter cover the largest contributors to the search error rate for both BDFT and SCD; the loading on each bin, the number of hash functions, the number of overloaded counters, and the overall design of the BDFT array. Also discussed is an analysis of SCD performance.

The complexity of BDFT requires that some assumptions be stated before further analysis of the algorithm can proceed. The bins in BDFT can be composed of any type of

data structure that supports the basic operations of insertion, removal and searching of items based on a flow identifier. The analysis of BDFT in this section uses bins composed of counting Bloom filters. Further, we assume that except where stated otherwise the counting Bloom filters are implemented using three independent hash functions (k=3). As described in section 4.3.1 BDFT operates most efficiently when only fully established flows are added to the first bin, so we assume that only fully established flows are added to the bin in this section. The use of SCD for pre-filtering can reduce the processing requirements of BDFT by 95%, depending on network traffic characteristics (described in section 6.2). We also assume that the flow removal recommendations of section 4.3.1 are followed, so that there is only one insert and one removal per flow.

The performance of BDFT is governed by five basic choices; number of hashes, counter size, bin size, bin time range configuration, and the hash functions used. These parameters describe the internal configuration of BDFT, and therefore must be selected before implementing the BDFT algorithm. These parameters are further described in section 5.2. When choosing the parameters of BDFT the environment created by the network to be monitored must be taken into account.

The major external variable affecting BDFT's performance is the total number of active flows that must be tracked. The number of active flows is normally related to network bandwidth, but can also vary substantially with the traffic mix. For "clean" traffic with little to no attack or port scanning attempts, active and complete flows can make up 95% of the bandwidth on the link. This type of traffic would typically be seen from professional users of bandwidth, e.g., large companies and institutions. Regular Internet traffic mix typically has a much larger percentage of bandwidth comprised of port scanning and attack traffic. In the case of general Internet traffic the number of active and complete flows will be lower given the same bandwidth usage as "clean" traffic. See the comparison of the "clean" *n_12* and "not-so-clean" *c_04* traces in section 6.1. Only the number of active and complete flows needs to be considered as a factor in BDFT's performance when using a pre-filtering algorithm such as SCD; otherwise, all flows in the traffic mix must be considered.

BDFT's performance is also affected by the distribution of flow durations in the network.

The two traces described in section 6.1 highlight the differences in traffic mixes between the general Internet and institutional traffic. Please refer to Figures 6.1 and 6.2. The distribution of flow durations in the network determines how many flows will fall into a BDFT bin for a specific time range. To achieve an efficient BDFT configuration, the time ranges for the bins, and the bin sizes, should be chosen to account for the distribution of flow durations by keeping the ratio of active flows to bin size fairly constant. Changes in the distribution during operation have a relatively minor impact on BDFT, even if changes in the distribution are large, as will be shown in section 5.4. Given the minor impact on performance, the flow duration distribution should be used only for fine tuning BDFT performance.

Calculation of the total search error rate requires one further assumption; we assume that no counters are overloaded. This leads us to look at the probability of a false positive in a single bin and then extrapolate to a search involving all bins. Overloaded counters introduce another type of error that may or may not affect searching. There is no guarantee that searches will be affected by overloaded counters, so we ignore the affect of overloaded counters for search errors. In the design of a BDFT array the expected number of overloaded counters should be kept to less than one per bin, as discussed in section 5.2.4.

5.2 BDFT Parameters

Analysis of BDFT involves several different parameters which affect algorithm performance as defined in section 5.1. In this section we show mathematically the expected performance of BDFT based on each of the parameters, taken independently of the others. Independent analysis of BDFT parameters leads us to present a series of basic heuristics for choosing a full set of BDFT parameters, given the goals of the algorithm designer. We choose a heuristic approach due to the BDFT's ability to adapt to a wide range of design goals and network traffic characteristics, based on the decisions of the designer. The large number of parameters in BDFT means that multi-dimensional graphs are required to describe the full range of interaction of many of the parameters. Figure 5.1 shows the search error

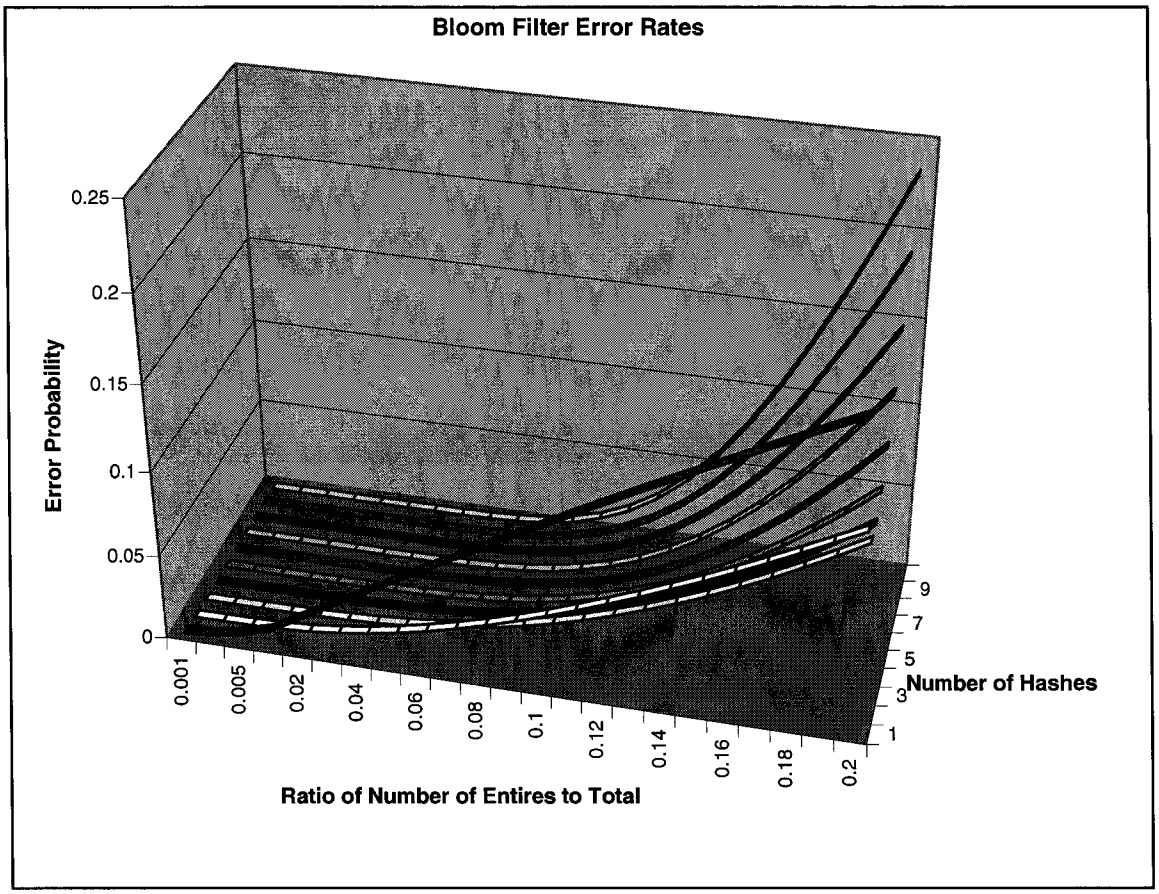

Figure 5.1: Probability of a false positive when varying the number of hash functions (equation 5.3)

performance of a Bloom filter when the ratio of entries to total entries (n/m as defined below) and the number of hash functions is varied. Figure 5.1 is computed from equation 5.3. In most sections a simple two dimensional graph is presented which shows the overall shape of the error curve with respect to the parameter being discussed. The heuristic approach presented is built on top of the analysis of performance when individual parameters are varied, and allows the designer to choose the best overall algorithm performance when their specific performance requirements are known.

In the following sections, multiple properties of Bloom filters, counting Bloom filters, and BDFT are described. These properties have a common nomenclature as described here:

- m - The size of the Bloom filter (in total entries).
- n - The expected number of entries in the filter. Also, the number of items in the set.
- k - The number of independent hash functions per Bloom filter.
- c - The maximum count in an entry of a counting Bloom filter.
- *item* - A member of a set that is stored in one or more *cells* in a Bloom filter.
- *cell* - One location in a filter, this can be a bit, or a counter, or in d-left hashing can contain a whole *item.*
- *bucket* - In d-left hashing a *bucket* contains a fixed number of *cells.*

5.2.1 General Guidelines for BDFT design

The following sections present specific guidelines and error rates for the design of a BDFT filter. Maintaining high accuracy in BDFT requires some additional general guidelines that should be followed in all implementations. In this section, we list the guidelines that should be followed along with a brief description of the benefits of the guideline.

In section 4.3.1, techniques for enhanced insertion and removal of flows are presented, and this chapter assumes that these techniques are employed in the design of the BDFT array. Also presented in section 4.3.4, are additional techniques for reducing backplane bandwidth, which, although they do not affect error rates directly, should be employed in any BDFT implementation.

Insertion and removal of flows from the Bloom filters can be accomplished with many variations of removal start point, search start point, and increment/decrement policies. Some of these strategies may provide better performance depending on traffic characteristics.

As mentioned in section 4.3.3, the removal of flows based on a false positive can have an impact on the performance of BDFT. The long-term running performance of BDFT and the effect of false positive removals is not directly analyzed in this chapter, and is left for future

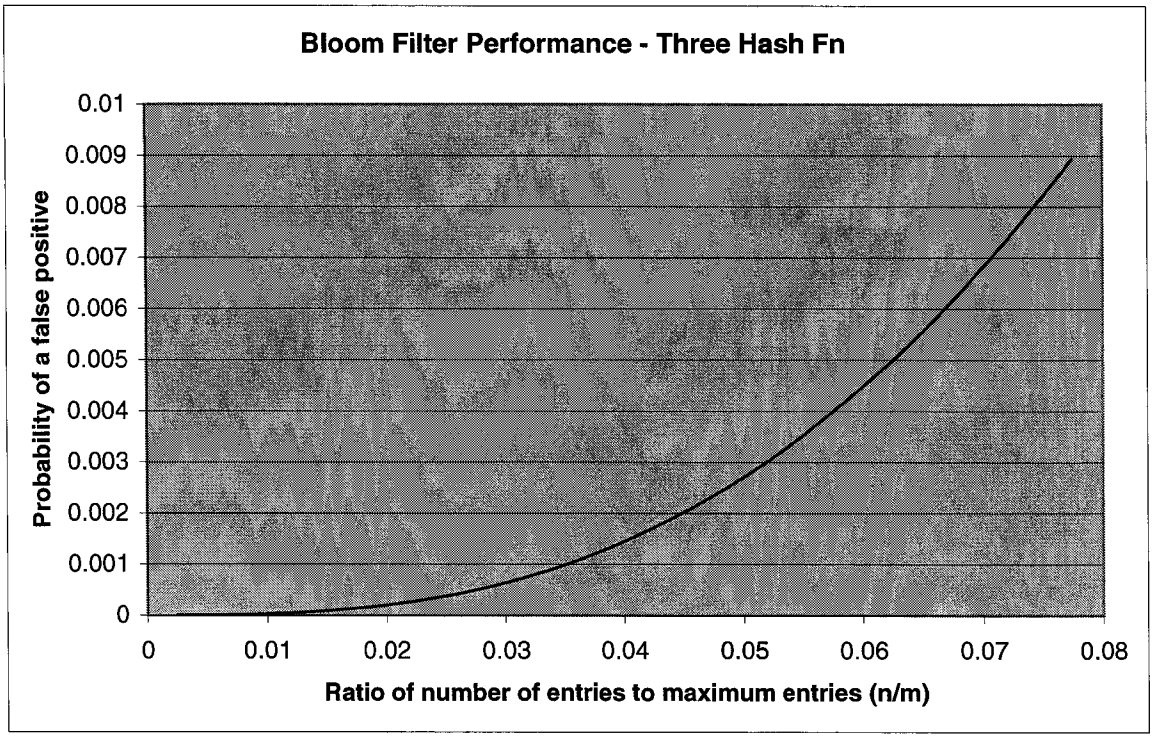

Figure 5.2: False positive rates in a Bloom filter with three hash functions (equation 5.3)

work. The effect of false positive removals can be reduced by implementing the techniques presented in section 4.3.3, and by choosing an appropriate starting search bin based on the BDFT array configuration and the traffic characteristics.

Finally, it is possible that some traffic monitoring applications would require that BDFT operate on sampled network traffic or sampled SYN/FIN/RST traffic. The effects of sampling on BDFT error rates are not analyzed in this chapter.

5.2.2 Bin Size and False Positive Rates in Bloom Filters

Bloom filters derive their remarkable space and time efficiency from a simple compromise; Bloom filters are not an exact data structure, they have a small probability of returning a false positive. A false positive occurs when an item which is not a member of the set represented by a Bloom filter is checked for membership in the filter and the filter returns that the item is a member. This situation occurs when other entries in the filter have set

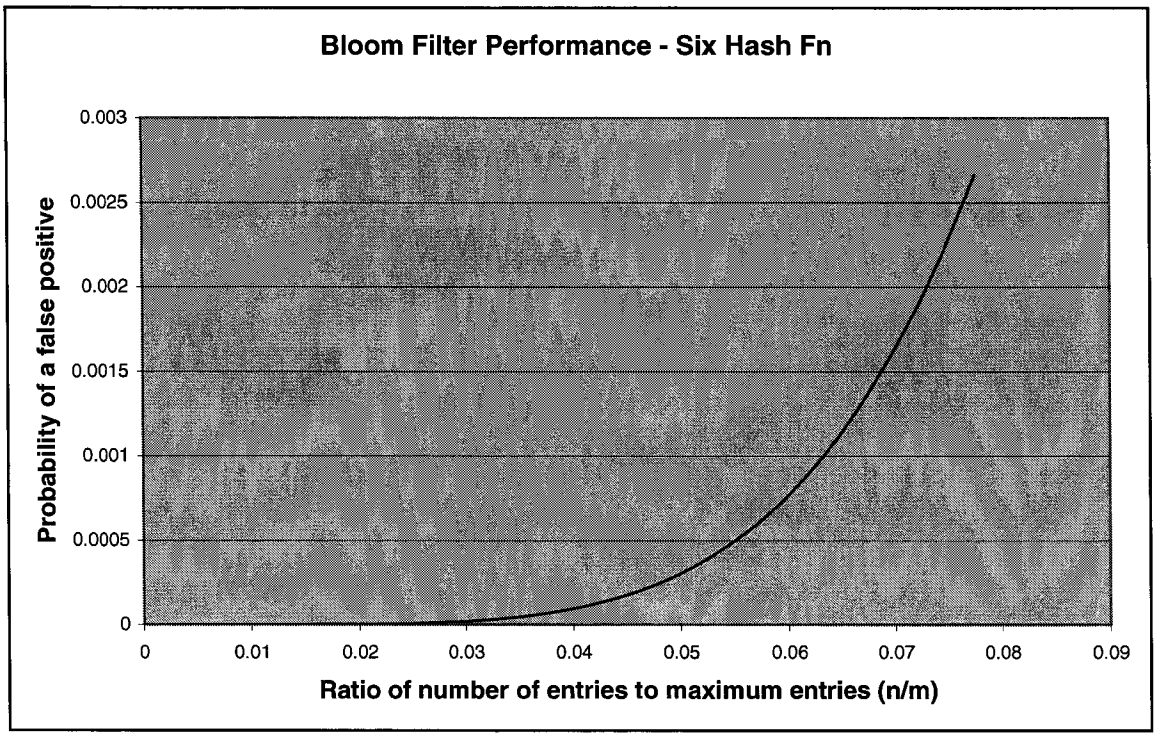

Figure 5.3: False positive rates in a Bloom filter with six hash functions (equation 5.3)

all the bits to true which correspond to the item which is not in the filter. For example, consider a filter with k=3, and the following example items in the filter with hashes;

1. { 200, 300, 400 }
2. { 250, 350, 450 }
3. { 1000, 2000, 3000 }

When the filter is queried for an item with hashes, i.e., { 200, 350, 3000 }, the filter will return that the item is a member of the set even though it is not. This occurs because the "false item" matches hash 200 from item 1, 350 from item 2, and 3000 from item 3. Note that a Bloom filter will never return a false negative, queries for an item that was added to the set will always be true.

The probability of a false positive can be calculated by examining the probability that a single bit in the filter is set to true (1). After n items have been inserted into a filter of

size m with k hash functions in the worst case there will be (refer back to section 5.2 for terminology);

$$\{n * k | n * k < m\} \tag{5.1}$$

filter entries set to true. The probability of a bit being false (0) is given by;

$$\mathrm{P[cell == 0]} = \left(1 - \frac{1}{m}\right)^{nk} \tag{5.2}$$

The probability of a false positive is determined by by the probability of selecting k bits that are set to true;

$$\mathrm{P[False\ Positive]} = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k} \tag{5.3}$$

Figures 5.2 and 5.3 show the effect of the loading (n/m ratio) on the false positive rate. Due to the exponential growth of the error rate it is desirable to operate Bloom filters with n/m ratios of 0.04-0.05 when high accuracy is required. The effect of the number of hash functions and calculating the optimal value of k for a given m and n is shown in section 5.2.3.

5.2.3 Number of Hash Functions

Bloom filters operate by utilizing a fixed number of independent hash functions, hereafter referred to as k, which are used to select the locations in the filter where the each inserted element is stored. The number of hash functions is a critical parameter which affects the performance of both the false positive rate and the number of overflowed counters within a BDFT bin. A high-level view when choosing the parameter k can be explained as a balance between competing forces. A Bloom filter with a specified m/n ratio has an ideal k such that;

$$k = \ln 2 \bullet \left(\frac{m}{n}\right) \tag{5.4}$$

Where k is rounded to the closest integer [21]. This criterion is balanced against the rise in the number of overflowed counters as k increases, as described in section 5.2.4. Also the computing requirements increase as k increases, due to the requirement to calculate the k hash values for every packet (unless the hash functions are implemented in hardware).

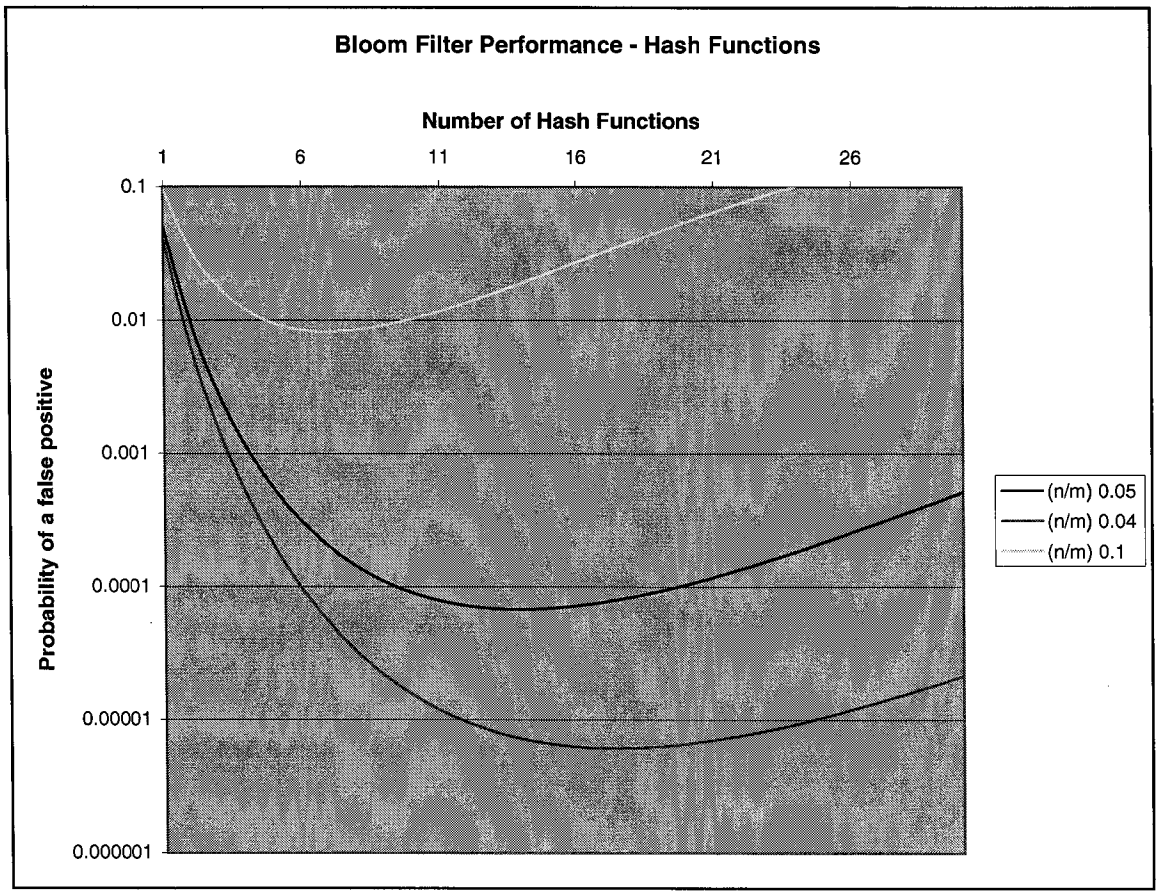

Figure 5.4: Change in false positive rates in a Bloom filter with change in k (equation 5.3)

Figures 5.2 and 5.3 show the error curves when three and six hash functions are used respectively.

An additional complication in choosing the number of independent hash functions is encountered when one considers a different number of hash functions for each bin in BDFT. Depending on the implementation and other parameters chosen varying the number of hash functions on a per-bin basis may lead to substantial performance improvements, since the n/m will be different for each bin.

For all of the reasons listed in this section the parameter k should normally be the last parameter chosen after all other parameters and variables have been decided upon. In other words k can be used in the last stages of designing a BDFT array to fine tune the error

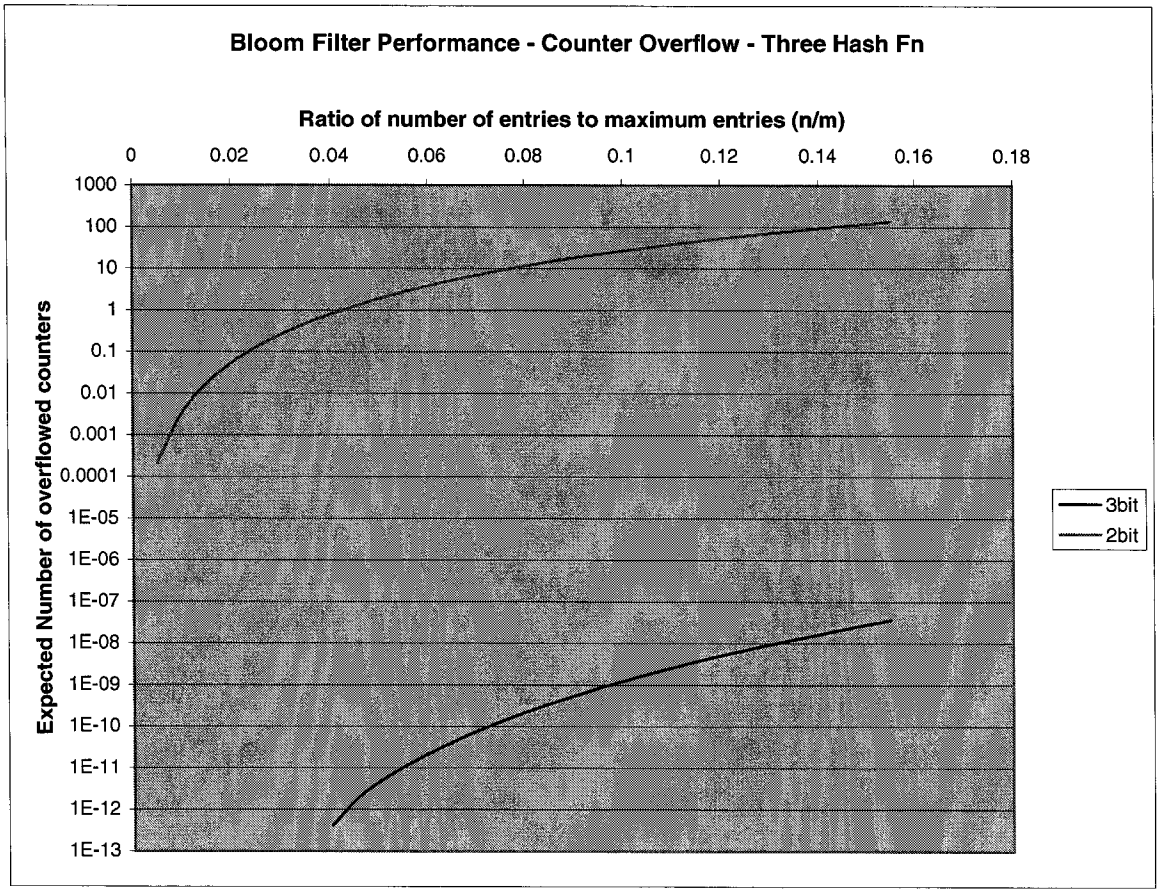

Figure 5.5: Counter overflow rates in a Bloom filter with three hash functions (equation 5.5)

rate vs. computational performance requirements. Figures 5.4 and 5.1 show the effect of various numbers of hash functions and n/m ratios on the performance of the filter, and can be referred to when fine tuning BDFT performance.

5.2.4 Counter Sizes and Overflow

BDFT requires a data structure that supports, the insert of, searching for, and removal of items. The standard Bloom filter with one bit (true/false) per entry does not support removal of items without generating false negatives. False negatives can occur in a standard Bloom filter due to the following property. A normal insertion can occur such that when two items are inserted into a standard Bloom filter the two items share a common hash.

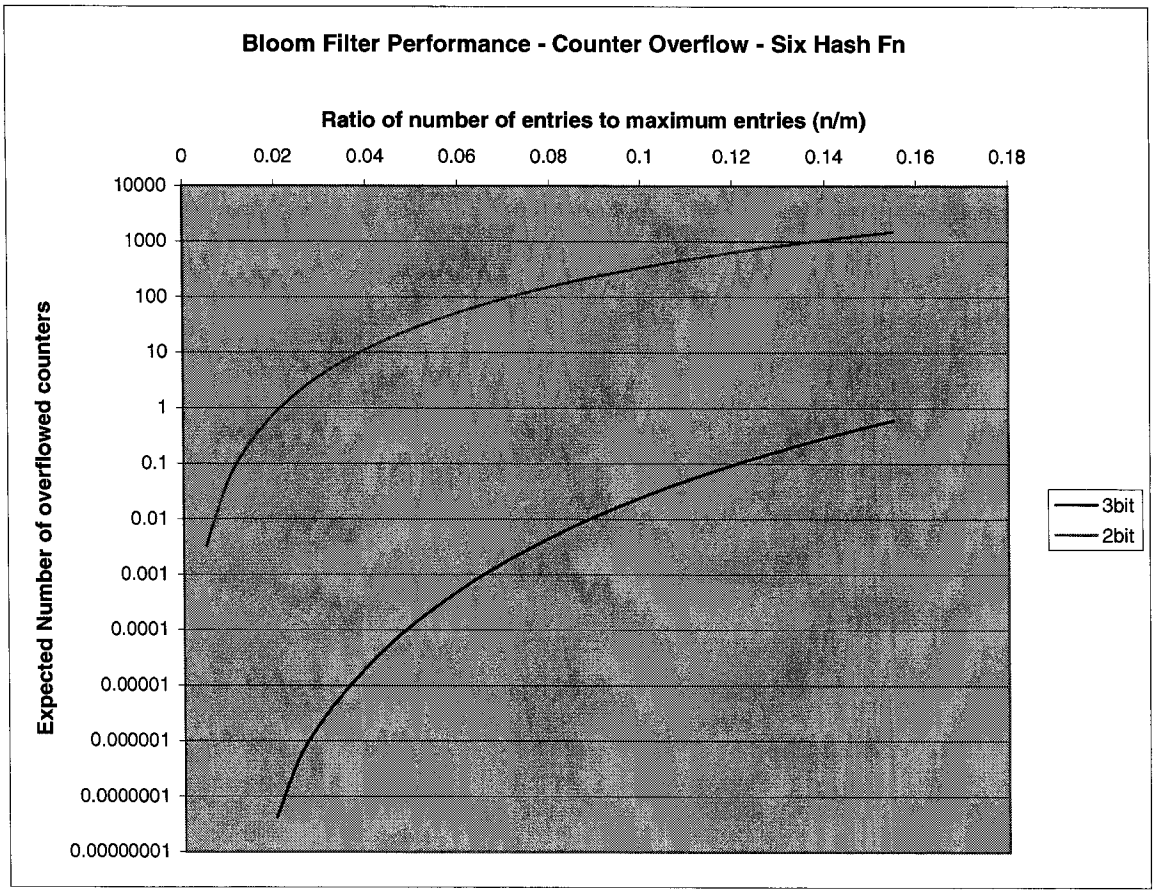

Figure 5.6: Counter overflow rates in a Bloom filter with six hash functions (equation 5.5)

This overlap of hashes will result in normal operation as all $(k * 2) - 1$ (k*2 due to two inserted items, -1 due to the shared hash) locations in the filter are set to true, and searches for these items will return true. However, if one of the items is removed, by setting it's k hash locations to false, then the shared hash will be set to false, and subsequent queries on the item still in the filter will return a false negative.

Counting Bloom filters extend standard Bloom filters to have a small counter for each filter entry instead of a single true/false bit [24]. When an item is inserted in the filter the counters corresponding to its hashes are incremented. If two items share a common hash value then the value of the counter corresponding to the common hash will be two. Removal of items from a counting Bloom filter decrements the counters corresponding to

the item's hashes. In the case of the shared hash the entry is decremented to one, and therefore, queries for the other item will still return true.

False negatives can still occur with counting Bloom filters if any of the counters in the filter "overflow". For a counting Bloom filter to maintain the properties of a Bloom filter, counters which are already at their maximum value when an item is inserted can not be incremented further. Further increments would cause an overflow to zero, which would break the properties of Bloom filters. Once an insert attempt is blocked from incrementing one or more of its counters due to overflow prevention, the counting Bloom filter will have a false negative due to the following situation. When all except one of the items corresponding to the overflowed counter have been removed the counter will be at zero since it was not incremented for one of the flows. At this point, queries for the remaining flow in the filter will return a false negative.

The expected number of overloaded counters can be calculated as follows. Each entry in a Bloom filter can be considered independently when determining the probability that it will be incremented. To cause an overflow the counter must be selected $c + 1$ times out of n inserts (where c is defined as the maximum count in a counter before overflow in section 5.2). We are interested in the probability of any overflow of the counter, as counter can overflow multiple times, defined as $c + 1$, $c + 2$, $c + 3$ ... $c + \infty$. The probability of having x matches in n inserts follows the binomial distribution if the counts are assumed to be independent. For a given maximum count c, the probability of overflowing a single counter/entry is given by;

$$\text{P[Counter Overflow]} = 1 - \sum_{x=0}^{c} \left[\binom{n}{x} \bullet \left(\frac{k}{m}\right)^{x} \bullet \left(1 - \frac{k}{m}\right)^{n-x} \right] \tag{5.5}$$

The number of expected overflows in a filter is then m * P[Counter Overflow]. Figures 5.5 and 5.6 show the expected number of overflowed counters in a filter with 100,000 total entries. The figures demonstrate the dramatic effect of adding a single bit to the size of the counters. This effect is shown by the 100,000X + difference in error rates between two and three bit counters for a variety of n/m loadings.

Overflow errors are a major factor when designing a BDFT array, as they disproportionally affect longer duration flows. This can be shown by considering that longer duration flows will be the last ones removed from BDFT. When a series of flows are added to BDFT within a short time all the flows land in the same bin, and if they share a common hash the counter corresponding to the hash value can overflow. Once an overflow occurs the removal of the first c flows to expire will proceed normally. In other words, the shorter duration flows will be tracked normally. The last flow still in the filter is the longest duration flow, but it is now a false negative after the removal of the shorter duration flows. The main purpose of BDFT is to track long duration flows, so overflow errors must be minimized as much as possible in any filter design, in the ideal case the expected number of overflowed counters should be less than one.

5.2.5 Bin Time Ranges and Sizing

One of the most complex decisions when designing a BDFT array is choosing the bin durations and sizes. This is also the most flexible aspect of the design, as BDFT can be customized to give very fine-grained or coarse grained feedback on the duration of flows. In section 5.4, we will present examples of a coarse-grained configuration which determines if the flow is short, medium, or long duration, and a fine-grained configuration where the approximate flow duration is returned by BDFT. This flexibility is derived from the selection of parameters for the bins. Each bin in BDFT tracks flows over a specified length of time and therefore has some simple parameters and performance guidelines. The parameters of each bin are those presented in sections 5.2.4, 5.2.2, 5.2.3, and this section. We also present search error rate guidelines take into account several factors, expected basic search error rates, number of overflows, and the distribution of flow durations in network traffic.

Basic search error describes the worst case expected error rate when searching for a specific flow in the BDFT array. This rate is dependent on the expected error rates for all of the individual bins. As mentioned in section 4.2, queries for a specific flow are performed by searching each bin individually, starting from the longest duration bin down to some reasonable minimum bin. Each bin has a associated false positive error rate, P[False

Positive in Bin x], based on the ratio of entries n/m and the number of hash functions k (equation 5.3). The worst case search occurs when the desired flow is in the last bin to be searched. To calculate the worst-case total error rate the success rates of all the searched bins are multiplied together. For example a BDFT array with 10 filters of which 8 are searched;

$$\text{P[Search Error Rate]} = 1 - \left(\prod_{x=10}^{2} (1 - \text{P[False Positive in Bin x]}) \right) \tag{5.6}$$

The accumulation of error when searching all of the bins, shown in equation 5.6, must be taken into account when designing a BDFT array. As a result, the search error for long-duration bins should be kept to a minimum, at least 99.9%+ accuracy is desirable.

The other major source of bin-specific error is the number of overflowed counters. The number of expected overflowed counters per bin should be kept to a minimum as described in section 5.2.4. This means keeping the n/m ratio and number of hash functions balanced with the size of the counters in the filter. Typical counter sizes are either 3 or 4 bits for heavily used filters, with 4 bits ensuring a very low probability of overflowed counters.

Once the parameters of the bins have been selected the next step is determining the flow duration information that should be tracked. If a fine grained measure of flow duration is required, then many bins with short time ranges can be used. In other applications, it may only be required to classify certain flows as being "long duration", for example, any flows over 2 minutes. For a coarse-grained analysis a simple BDFT array design using three bins is all that is required.

The *prime consideration* in choosing bin durations is the aging process for bins. During aging all bins that need to be aged should expire at the same time, which allows an efficient implementation of the aging algorithm by simply copying or updating pointers to bins that have just expired from longer duration to shorter duration bins. Having all the bins expire at the same time can be achieved by choosing bin durations that are a multiple of the shortest bin duration, and a multiple of the next lowest bin duration. For example, the three bin setup for coarse grained monitoring could use bins with durations of 20 seconds, 80 seconds, and infinite seconds. For fine grained monitoring many bins with short duration

can be selected, i.e., 20, 20, 20, 40, 40, 40, 80, 80, 80, and etc.

Determining the required size of each bin is the final and most crucial step in the design of a BDFT array. The sizing of the bins depends on the flow duration distribution that is expected in the network. The expected distribution does not need to be exact, but should provide a general guideline for the number of flows expected in each bin. For example, the flow duration distribution described in section 6.1 shows that 75% of valid flows last less than 15 seconds (also see section 6.1 for more information on flow duration distribution). For a network where 100,000 valid flows are expected to be active at any time, a maximum of 75,000 would be expected to be stored in a bin containing flows from 0-15 seconds. This bin would need to be sized accordingly to reduce expected error rates. Higher duration bins typically have fewer flows stored, and therefore can be sized smaller to reduce memory usage. Bin sizes should be a multiple of a base size to increase the efficiency of the aging algorithm.

Section 5.4 describes the full design of a BDFT array with SCD and FIN/RST filtering.

5.3 SCD Parameters and Performance

Dual-Filter SCD (chapter 3) uses standard Bloom filters to track the state of TCP connections, so the parameters of the Bloom filters form the the first set of parameters of SCD, as described in section 5.2. Namely, m, n, and k; the maximum size of the filter in entries, the actual number of entries in each filter, and the number of independent hash functions, respectively. These are the main parameters in determining the false positive rate (equation 5.3), and therefore the accuracy of Dual-Filter SCD. The false positive rate is the primary consideration in SCD filter design.

Other parameters that must be considered when designing a Dual-Filter SCD filter are related to the number of Bloom filters per direction of traffic flow, and the lower bound of the maximum connection time. In the normal Dual-Filter SCD configuration there are two filters per direction. Increasing the number of filters to three or four per direction may result in a minor increase in accuracy at the expense of increased computational requirements. In

this analysis we will only consider the case of two filters per direction.

The Bloom filter sizes in SCD should be matched to the number of flows created per second. Both valid and invalid flows must be considered, including SYN packets which are part of port scanning, DoS attacks, and all other SYN packets. Typical network traffic has the SYN flag set in about 5% of the total packets. For example, on a fully utilized 1 Gbps link processing about one million packets per second (125 bytes/packet on average), there will be about 50,000 (5%) SYN packets per second. Of the 50,000 SYN packets it can be roughly assumed that they will be split evenly between the two directions in SCD (direction is defined in section 3.1), so there will be 25,000 new flows per second per direction in the worst case, assuming that every SYN is part of a new flow. Once the average number of new flows per second has been estimated the next step in designing an SCD filter is choosing an acceptable lower bound of the maximum connection time (also defined in section 3.1).

The maximum connection time and lower bound of the maximum connection time are related in Dual-Filter SCD, the lower bound is half the maximum connection time. Selection of the maximum connection time parameter allows several factors to be balanced:

1. The maximum time a TCP connection is expected to take to complete (preferably select the 99th or higher percentile of all connection times).

2. The average number of new flows per second.

3. The desired memory usage of Dual-Filter SCD.

In the above example there were an average of 25,000 new flows per second per direction. Assuming that 99.7% of TCP flows in the network complete the connection handshake within an average of 12 seconds, the lower bound of the maximum connection time is chosen to be 8 seconds, making the maximum connection time 16 seconds. The total number of flows that one filter must hold is therefore 8 * 25,000 = 200,000. To size a Bloom filter such that there is a 1% expected false positive rate, the values for the expected number of flows, number of hash functions, and false positive rate can be substituted into equation 5.3. In this example, a filter size of approximately 2,000,000 entries is required, leading to a memory usage of 2,000,000 * 4 filters * 0.125 bytes/bit = 1,000,000 bytes.

5.4 Putting it all together

In this section, we apply the techniques discussed in the previous sections of this chapter and chapter 4 to design BDFT filters in two example cases. We choose two design goals that represent possible real-world scenarios for flow duration. The first design requires a general flow duration tracker that is capable of determining a flow's duration with 50%-80% accuracy from 60 minutes down to 45 seconds. The second design is focused on simply classifying flows as long duration, which is defined as a flow duration longer than two minutes. Using BDFT as a classifier in the second design allows the long duration flows to be flagged for further processing by a deep packet inspection device to determine if they are P2P traffic or other traffic types. For these filter designs the network traffic is assumed to approximately follow the flow duration distribution presented for the *C_04* trace in section 6.1. These filter designs and accuracy calculations utilize the methods for enhanced insertion and removal of flows, as discussed in section 4.3.1.

The design of a BDFT array involves balancing the many parameters that affect search performance, memory usage, and computational requirements. The primary parameters in the design of a BDFT array are the number of flows that are expected to be in each bin and the size of the bins, as this ratio determines the loading of a particular bin. As a general guideline, to design a BDFT array with a high search accuracy means that the long duration bins should have a search accuracy of at least 99.9+%. For the fine-grained design the sizing of the long duration bins can therefore begin with the dual requirements of at least 99.9% accuracy and at least 50% time accuracy.

The number of flows in each bin can be estimated from the flow duration distribution. We determine the number of flows in each bin according to the following assumptions:

1. There are 1,000,000 terminating flows per hour.

2. Flows arrive at a uniform rate.

3. The flow duration distribution is uniformly distributed within a bin.

Table 5.1: Expected active flows in bins (fine-grained design)

Bin Start Time (sec)	Bin End Time (sec)	Active Flows Ending in Bin	Arrival Rate (Longer Dur)	Active Flows in Bin
0	15	1020.59	73.66	2125.48
15	45	517.33	39.17	1692.45
45	75	245.43	22.81	929.69
75	105	119.00	14.88	565.26
105	165	272.35	5.80	620.18
165	225	46.57	4.24	301.25
225	285	25.92	3.38	228.76
285	405	82.06	2.01	323.63
405	525	51.99	1.15	189.59
525	765	66.32	0.59	208.90
765	1245	71.95	0.29	213.22
1245	2205	99.05	0.09	183.48
2205	3165	33.34	0.02	51.09
3165	3600	8.87	0.02	26.62

From assumption 3 the average termination time of flows in a bin is half the bin duration (Note that assumption 3 is worst-case, in normal traffic the distribution is skewed towards shorter duration flows, resulting in fewer flows in the bin). The number of flows that terminate in a given bin within a certain time can be directly read from the flow duration distribution. Also the flows that are longer duration than the current bin, but are passing through the bin must be taken into account. Given the above assumptions, the number of flows active in bin b is given by Little's equation, where the Average Time in Bin is half of the bin duration:

$$\text{Flows Ending in Bin} = \text{Average Time in Bin} \bullet \text{Arrival Rate (ending in bin)} \tag{5.7}$$

Where the arrival rate is calculated from the flow duration distribution by the formula:

$$\text{Arrival Rate (ending in bin)} = \frac{\text{1,000,000 flows/hour} \bullet \text{P[Flow Terminating in the Bin]}}{\text{3600 seconds/hour}} \tag{5.8}$$

Table 5.2: Expected active flows in bins (coarse-grained design)

Start Time (sec)	End Time (sec)	Active Flows Ending in Bin	Arrival Rate (Longer Dur)	Active Flows in Bin
0	20	1445.20	61.00	2665.15
20	110	1433.28	13.22	2623.23
110	3610	2346.73	0.00	2346.73

Accounting for the contributions of longer duration flows to the flows in the bin requires summing all the flows which are in the bin but will not terminate here, and instead are moving to higher duration bins. Little's equation still applies, but the Average Time in Bin is now the full bin duration. Let n represent the number of bins and c be the current bin;

$$\text{Arrival Rate (not ending in bin)} = \frac{\text{1,000,000 flows/hour} \bullet \sum_{i=c}^{n} \text{P[Flow ends in Bin i]}}{\text{3600 seconds/hour}} \tag{5.9}$$

Table 5.1 shows an example configuration of bin time ranges which match the requirements of the fine-grained design. The number of active flows expected in each bin was calculated using equations 5.7, 5.8, and 5.9. Table 5.2 gives an example configuration for the coarse-grained design. The number of active flows in each bin was calculated with slight modifications to assumption #3 for the very long bin durations, to skew the average length of time for each flow in a bin to shorter durations.

Once the bin time ranges and number of active flows in each bin has been determined the bins can be sized. Heuristics for sizing the bins are as follows:

1. A short duration (10-15sec) medium accuracy bin (95%) should be used to filter the large number of very short duration flows.

2. The short duration bins will not be searched in many cases, and therefore can be designed to have a higher false probability error rate in order to save on memory.

3. The long duration bins should have high accuracy (99.9%) for good search accuracy.

Table 5.3: Example BDFT array configuration (coarse-grained design)

Start	End	Max Entries	# Flows	P[false +]	# overflows	# Hash	# Bits
0	20	131072	2665	0.00021	0.0720	3	2
20	110	131072	2623	0.00020	0.0676	3	2
110	3610	131072	2346	0.00014	0.0435	3	2

Table 5.4: Example BDFT array configuration (fine-grained design)

Start	End	Max Entries	# Flows	P[false +]	# overflows	# Hash	# Bits
0	15	131072	2125	0.00011	0.0294	3	2
15	45	131072	1692	5.5E-05	0.0119	3	2
45	75	65536	929	7.2E-05	0.0086	3	2
75	105	65536	565	1.7E-05	0.0012	3	2
105	165	65536	620	2.2E-05	0.0017	3	2
165	225	32768	301	2E-05	0.0008	3	2
225	285	32768	228	8.8E-06	0.0003	3	2
285	405	32768	323	2.5E-05	0.0010	3	2
405	525	32768	189	5E-06	0.0001	3	2
525	765	32768	208	6.7E-06	0.0002	3	2
765	1245	32768	213	7.2E-06	0.0002	3	2
1245	2205	32768	183	4.6E-06	0.0001	3	2
2205	3165	16384	51	8E-07	0.0000	3	2
3165	3600	16384	26	1.1E-07	0.0000	3	2

4. The number of flows normally decreases logarithmically with the duration, so longer duration bins should cover more time and be reduced in size.

5. The binning process introduces inaccuracy when the actual duration of the flow is not the average duration of the bin, so longer duration bins should cover more time, and the shorter duration bins less time. This keeps the relative inaccuracy involved in the binning process to a minimum.

Table 5.5: Example BDFT array theoretical performance

Design	Memory Usage (bytes)	Accuracy	Expected # Overflows
Fine-Grained	180224	99.96%	0.055
Coarse-Grained	98304	99.95%	0.183

Example configurations which follow these heuristics are shown in Tables 5.3, and 5.4. The performance of the fine-grained and coarse-grained designs is shown in Table 5.5.

5.5 Computational Requirements of BDFT

The computational requirements of BDFT can be calculated through analysis of the basic BDFT operations, insert, remove, search, and aging. In this section we analyze the performance of these operations and draw comparisons with another data structure that has been suggested for stateful network flow tracking, d-left hashing [34] (d-left was introduced in section 2.4.3). For this performance analysis it is assumed that counting Bloom filters are used for the BDFT bins, although for some operations there may be other data structures such as space-code Bloom filters that would be more efficient. Note that for both d-left hashing and counting Bloom filters the first step in every operation is to calculate the hashes for the item, since this step is common to all operations it will not be included in this analysis. For d-left hashing it is assumed that four hash functions are used and there are six cells per bucket (see section 5.2 for terminology). For d-left hashing there is no suggested way to store the cells in the buckets. For example, an array with an active/inactive flag for each cell, or a doubly-linked list would work, but both carry additional computational requirements not discussed in the d-left paper. D-left hashing implemented with the array based scheme requires checking every cell on every operation, so the worst case performance discussed below becomes the average performance, and maintaining a doubly-linked list requires extra pointer operations. BDFT has one further general advantage over d-left hashing, the computational requirements for scale in "constant time" vs. the number of flows, whereas d-left scales logarithmically.

Insertion of elements into BDFT requires modification of the first bin only. The current counters corresponding to the items hashes must be read, incremented, and then updated with the new values. This process requires k memory reads and k writes. D-left hashing requires that the buckets corresponding to the hashes be searched for the incoming flow, at worst this is 6*4 = 24 reads and compares (in our example with six cells/bucket). Once the flow is confirmed not to be present the flow must be added to the least-filled bucket with tie-breaks to the left, requiring another four comparisons and a memory write. For both d-left hashing and BDFT, the memory reads can be parallelized in hardware.

Removal of items from BDFT requires searching the bins starting with the shortest duration bin first. Given a nominal BDFT bin setup for fine-grained monitoring and nominal Internet traffic, about 75% of searches will end with the first bin, about 15% will end in the second, and some flows will require searching all bins. Searching one bin requires k memory reads, and k comparisons. The removal operation is the reverse of insertion; decrement the counters and write the new values to the filter. D-left hashing must search 6*4=24 buckets in the worst case to find the flow, requiring 24 memory reads and comparisons. Once the flow is found, d-left must update the doubly-linked list or the flag the cell inactive.

Searching for an item in BDFT starts with the longest duration bin and works down to shorter duration bins, based on the assumption that the longer duration bins are the least likely to generate a false positive. Searching is the most expensive operation in BDFT, with the worst-case requiring reading the counters corresponding to the hashes in every bin except the shortest. This would require k * (number of bins - 1) memory reads and the same number of comparisons. The requirements for d-left hashing are the same as insertion and removal, 24 reads and comparisons.

Aging of bins is one of the more expensive operations that must be performed during BDFT operation. Depending on the implementation, aging bins may require merging counting Bloom filters of different sizes, without overwriting the data in the recipient filter. In this case, the merging process requires reading all of the entries in both filters, adding them together, and writing the results to the longer-duration filter. In special cases the aging process can be optimized. For instance, if both filters are the same size, and the recipient

filter can be overwritten (because its contents were just aged to a longer duration bin), then aging the filter is a simple pointer update. For d-left hashing with state maintenance, every cell in the buckets must be read, a decision or comparison made, and its state updated. This occurs every 15 seconds, or as often as whatever the minimum duration state represents. In a nominal BDFT configuration only the shortest duration filter needs to be processed every 15 sec, and the second shortest duration filter every 30 sec, and so on for all filters. The longer duration filters need to be processed only every 5-10 min.

In summary, BDFT provides very good computational requirements in the average case. Most operations require lookups in only one bin. Searches are the only operation which require multiple bin searches in the average case.

Chapter 6

Experimental Analysis

This chapter describes the expected performance of BDFT and SCD when deployed in real-world situations. Internet traffic traces are used to drive the experimental implementations of BDFT and SCD to obtain an indicator of expected performance. Section 6.1 describes the characteristics of the traces and shows that the traces we selected for performance analysis constitute realistic and diverse examples of Internet traffic. We also present the flow duration distribution for both traces, a metric that is rarely analyzed in most traffic analysis studies. Section 6.2 describes the performance of our implementation of Dual-Filter SCD. Section 6.3 describes the performance of BDFT and compares the performance of BDFT with several other possible flow duration tracking methods.

6.1 Experimental Trace Characteristics

To verify and validate the real-world performance of SCD and BDFT, we implemented an experimental test bed and used Internet traffic traces to drive the tests. We obtained traces of Internet traffic from the well-known networking research organizations CAIDA [38] and NLANR [39], with the two traces hereafter referred to as *C_04* (CAIDA) and *N_12* (NLANR). This section describes many of the intrinsic characteristics of these two traces and explains why they are representative of the diverse extremes of Internet traffic. In general, the *C_04* trace represents normal "dirty" public backbone Internet traffic, with many packets being invalid attempts at port scanning or DDoS attacks. This trace was

Table 6.1: Trace characteristics

	NLANR 2003-12 (*N_12*)	As a % of total	CAIDA 2003-04 (*C_04*)	As a % of total
Total Packets	196,956,306		202,510,985	
Total TCP Packets	56,992,573	28.94%	175,418,691	86.62%
Total Bytes	46,472,308,705		95,944,872,321	
Total TCP Bytes	41,482,633,988	89.26%	91,766,946,651	95.65%
Avg. Bandwidth (Mbit/s)	98.48		203.33	
Avg. Bytes Per TCP Pkt.	727.86		523.13	
Duration (seconds)	3600		3600	

collected by CAIDA from both directions of an OC48 link at AMES Internet Exchange (AIX) on Apr. 24, 2003, at Mountain View, CA, a west coast peering link for a large ISP [38]. The second trace, *N_12*, was obtained by NLANR in December of 2003, from their NCAR Gigabit tap (at the National Center for Atmospheric Research, Boulder) [39]. This trace represents the other end of the traffic spectrum from *C_04*, being fairly "clean" and containing a low number of active flows and very little or no attack and port scanning traffic.

Both traces represent one hour of Internet traffic. These long duration traces were selected to demonstrate the performance of BDFT over long periods of time. Table 6.1 shows the basic characteristics of each trace. At this very coarse-grained level of traffic analysis, both traces appear quite similar. The average bandwidth in both traces is roughly similar at 100Mbps and 200Mbps, sufficient to demonstrate the performance of BDFT and SCD on high speed links. The total number of packets is also very similar, however the number of TCP packets is over three times higher in the *C_04* trace. All other coarse-grained trace characteristics are quite similar between the two traces, including average bytes per TCP packet.

Table 6.2 shows the first striking difference between the *C_04* and *N_12* traces. The percentage of total packets which are TCP packets with one of the main control flags

Table 6.2: Trace characteristics for TCP control packets

	NLANR 2003-12 (*N_12*)	As a % of total	CAIDA 2003-04 (*C_04*)	As a % of total
Total Packets	196,956,306		202,510,985	
SYN	732,075	0.37%	15,608,680	7.71%
FIN	586,000	0.30%	6,084,826	3.00%
RST	52,628	0.03%	3,914,433	1.93%

Table 6.3: Trace characteristics for TCP 5-tuple flows

	NLANR 2003-12 (*N_12*)	As a % of total	CAIDA 2003-04 (*C_04*)	As a % of total
Total Flows	352,410		11,215,873	
Total Established Flows	274,473	77.88%	555,927	4.96%
Average Active Flows	11,284		901,245	
Timed Out Flows	430	0.16%	4376	0.78%
Unique IPs	97,036		2,681,172	

(SYN, FIN, RST) turned on is over ten times lower in the *N_12* trace. Another striking difference is the number of RST packets, the percentage of RST packets in the *C_04* trace is over sixty times higher than the *N_12* trace. Given that RST packets typically indicate abnormal connection termination, this large difference indicates that the *C_04* trace has many connections that do not follow normal TCP rules, such as port scanning or DoS attack traffic.

Table 6.3 is a fine-grained look at the flows in each of the traces, and therefore highlights the substantial differences between the two traces. The large percentage of SYN packets in the *C_04* trace can now be confirmed to be related to the large number of incomplete flows, over 95% in this case. In other words, out of 15M SYN packets sent, only approximately 1.5M resulted in fully established flows. Also, due to the large number of incomplete connection attempts (likely port scanning), there are a very high number of active flows on average compared to the *N_12* trace. For these reasons the *C_04* trace is considered to be

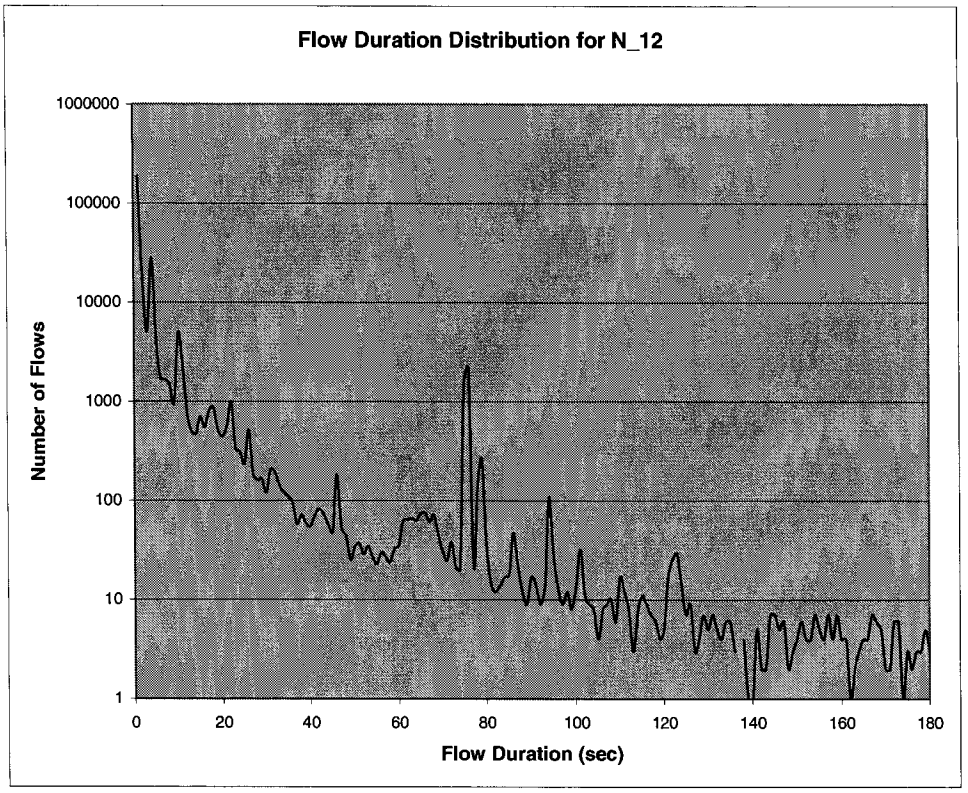

Figure 6.1: Flow duration distribution for trace N_12

a good example of "dirty" Internet traffic and the *N_12* trace is a good example of "clean" traffic (with almost 80% of flows being valid in the *N_12* trace).

The average number of active flows gives a rough estimate of the number of flows actively transmitting data in the trace. To calculate this metric, the flow timeouts are set to thirty seconds for flows that have contained at least one FIN or RST, and ten minutes for all other flows. Flows that began before the start of the trace, or have their connection phase span the end of the trace, are ignored by the flow tracker and are not included in the results.

The distribution of flow durations is an important factor in the design of a BDFT filter array, as described in section 5.4. The duration distribution allows a rough estimation of the required size of the bins in BDFT, and therefore an estimation of the total memory usage of the BDFT array. The distribution of flow durations is also an interesting measure of the characteristics of a network trace. However, duration distributions are almost never

presented in the literature, even in papers focused on trace analysis. One of the few papers to discuss the number flows that last specific lengths of time is "Dragonflies and Tortoises" [36], where they find that 75% of flows last less than two seconds. Knowledge of the duration distribution can also be exploited by algorithm designers to enhance the performance and memory requirements of network flow tracking. For example, BDFT makes use of the fact that most flows are short duration to reduce the memory required for longer duration bins.

Figure 6.1 shows the flow duration distribution for the *N_12* trace. Only flows that are fully established within the trace are counted. This distribution shows that 75% of fully established flows that are less two seconds long. This duration distribution is characterized by the sharp falloff in the number of flows as the duration increases. The large increases at certain durations, such as 80 seconds, indicate repeated transfers of specific sized data or use of specialized applications.

Figure 6.2 shows the flow duration distribution for the *C_04* trace. Like the *N_12* trace this duration distribution has a large number of flows that last less than two seconds, 50% in this case. However, unlike the *N_12* trace, the *C_04* trace has a heavy-tailed duration distribution, in keeping with the norms for general Internet traffic. Figure 6.3 shows the duration distribution over a longer period, further detailing the heavy-tailed nature of the durations in this trace.

6.2 Experimental Performance of SCD

Analysis and validation of SCD performance was accomplished by implementing SCD and running experiments based on Internet traffic traces. To verify the SCD results, a 100% accurate flow tracker was implemented which is able to track the connection status of every flow observed in the traces. This flow tracker was implemented using standard per-flow tracking and measurement techniques. We defined a flow to be the standard 5-tuple of IP source and destination address, TCP source and destination port, and protocol type.

To evaluate the results of our experiments, the connection status reported from the flow tracker was compared to the results returned from SCD, giving one of three results. First,

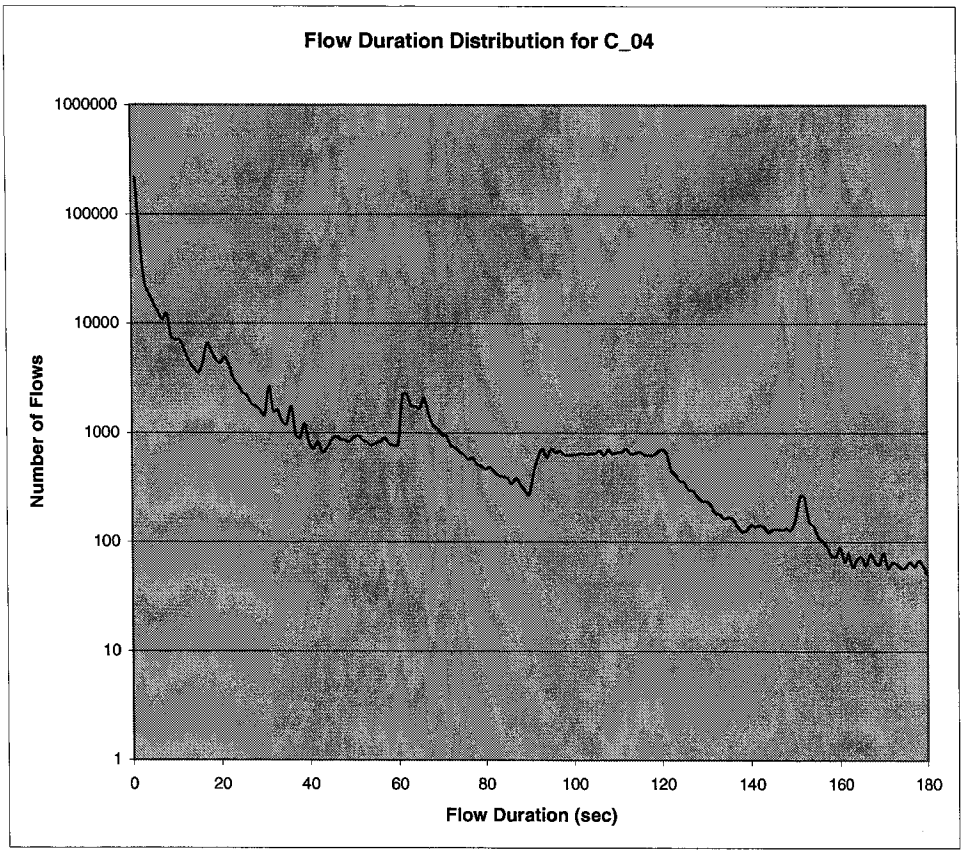

Figure 6.2: Flow duration distribution for trace C_04

the SCD algorithm can indicate that the flow is established, in agreement with the flow tracker, this counts as a successful indication by SCD. Second, SCD returns that the flow is established, but the flow tracker indicates that the flow is not established, this is a false positive and is a characteristic of Bloom filters as explained in section 3.1. Third, SCD can fail to report an established flow, and the flow tracker indicates that the flow is established, this is a false negative and occurs when the flow took longer to establish than the maximum flow connection time parameter of SCD.

The SCD algorithm was implemented in the C programming language using standard techniques of bitmap manipulation to implement the Bloom filters. Packets are read from the trace and the Bloom filter hashes are generated by three independent hash functions from the following packet header information; IP source and destination address, and TCP

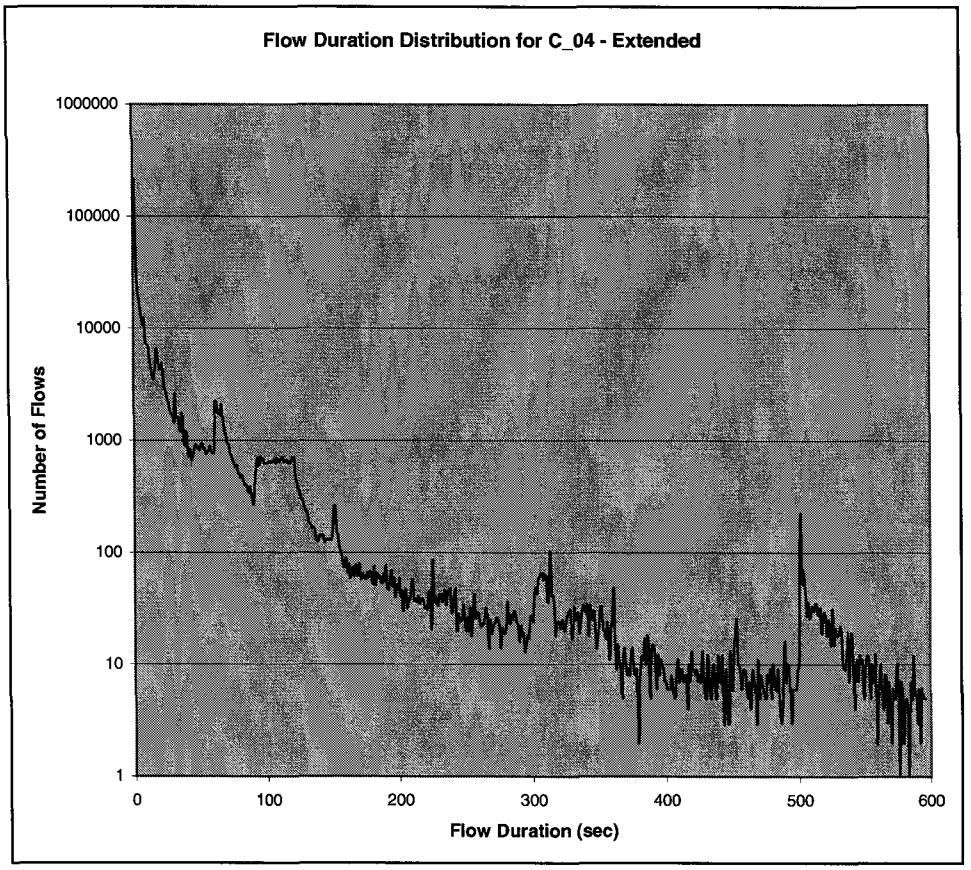

Figure 6.3: Flow duration distribution for trace C_04 extended to 600sec duration

source and destination port. The Dual-filter SCD method, described in section 3.2, was implemented by clearing the old filter for either direction and then simply updating pointers to exchange the new and old filters. For every SYN packet in the trace the hashes are passed to the SCD_Packet function, which returns true if the flow is now established and false if it is not. The connection establishment indication is stored and compared with the final state of the flow to determine if it was a successful indication.

For our simulations two one-hour long traces were selected which are representative of classic Internet traffic mixes. Out of 12.7 million flows in the *C_04* trace only 556,000, or 4.3%, are valid fully-established flows. This high invalid-flow ratio combined with the high number of active flows make this trace a good worst-case test of SCD. The second

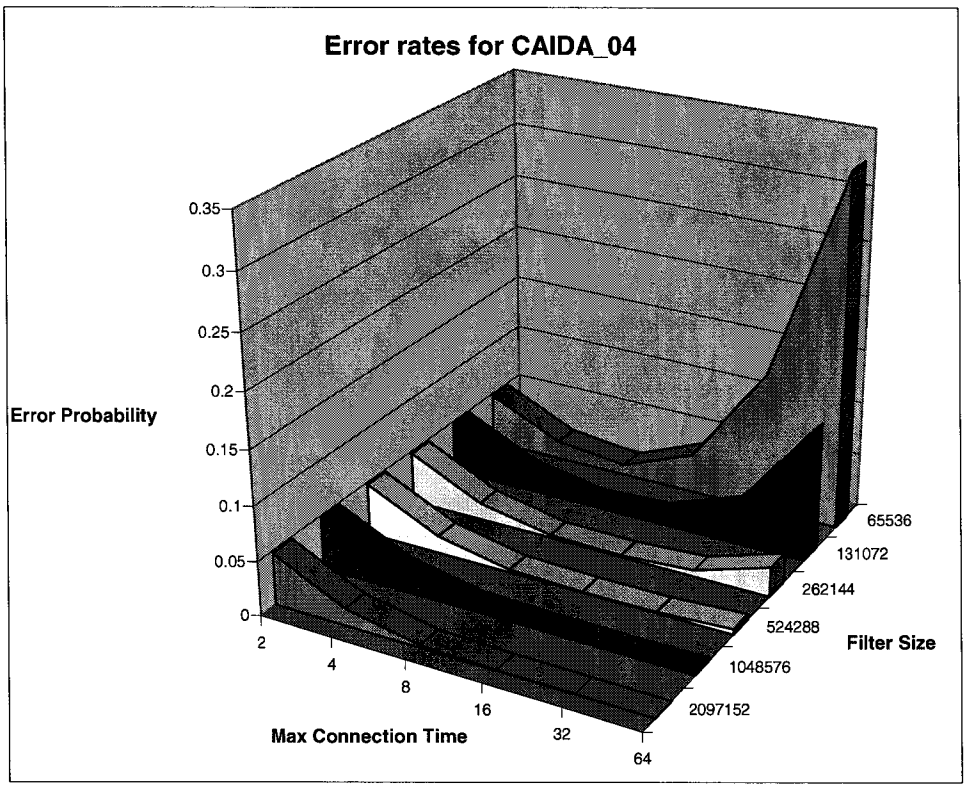

Figure 6.4: SCD performance for trace C_04

trace trace, *N_12*, represents the other end of the traffic spectrum, containing a low number of active traces and very little or no attack and port scanning traffic. Out of 358,048 flows in this trace 274,473 are valid (76.7%), making this trace an example of the best-case performance of SCD.

Our experimental results are presented in Figure 6.4, Figure 6.5, and Table 6.4. To evaluate the performance of SCD, we varied two parameters of the algorithm, filter size and maximum connection time. Filter size represents the size of one Bloom filter in bits. Given that dual-filter SCD uses four filters, the total memory usage can be calculated as:

$$\text{Memory Usage} = \left(\frac{\text{bits per filter}}{8 \text{ bits/byte}}\right) * 4 \tag{6.1}$$

The maximum connection time specifies the maximum time that can elapse between a SYN packet in one direction and the response SYN from the other direction. Due to the nature of

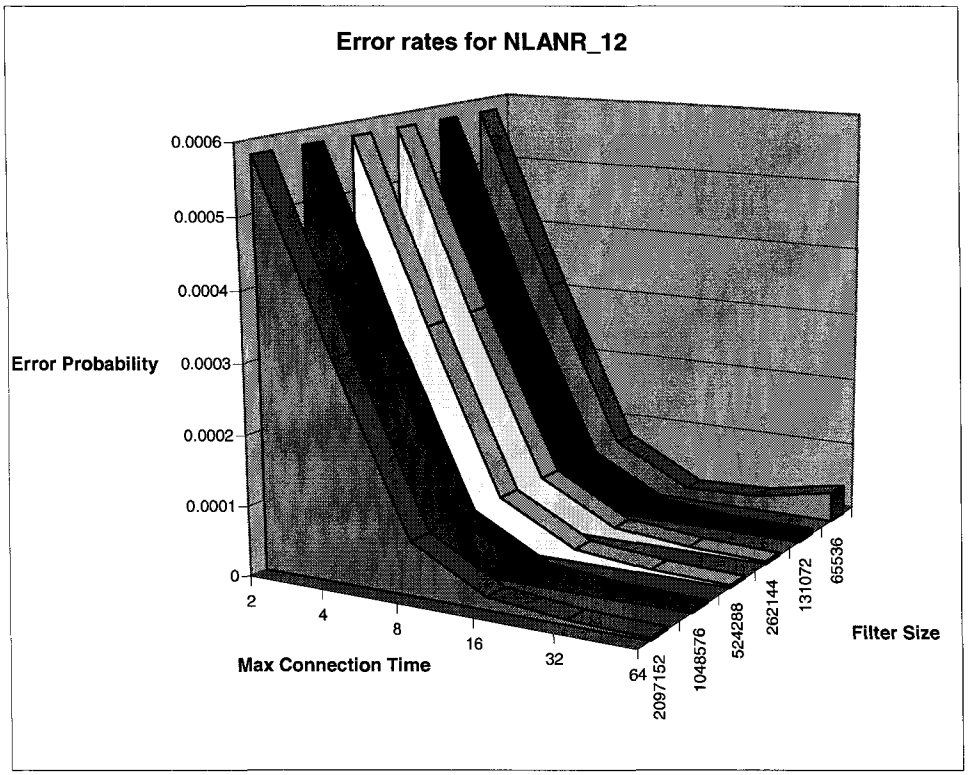

Figure 6.5: SCD performance for trace N_12

dual-filter SCD, the actual maximum connection time on a per-flow basis varies from Max Conn Time / 2 to Max Conn Time. If this time is exceeded a false negative is generated.

Table 6.4 lists example configurations and shows that SCD is 99%+ accurate even at only 32k bytes of memory usage. This level of memory usage indicates that SCD can be implemented using SRAM at the datapath level of a router or other network device. By increasing memory usage to 512k, over 99.9% accuracy can be achieved. As a comparison of memory usage, our per-flow tracker used 24MB of memory to store about one million flows, or forty-eight times more memory than the 99.9% accurate SCD.

An important observation can be made by analyzing the SCD results. Some of memory usage plots in Figure 6.4 have a local minimum, indicating that there is an ideal setting for the maximum connection time parameter. The ideal setting occurs when the false negative and false positive rates balance out. With a low maximum connection time setting

Table 6.4: SCD parameters and accuracy

Trace	*Memory(bytes)*	*Max.Conn.Time*	*Accuracy*
N_12	524288	64	99.9996%
N_12	32768	16	99.9982%
C_04	524288	32	99.9685%
C_04	32768	8	99.0553%

many flows fail to establish before the filters are cleared, leading to a high number of false negatives. As the maximum connection time parameter increases, the Bloom filters start to become overloaded with flows, leading to a high number of false positives. The increase in false positives is balanced by the decrease the number of false negatives as maximum connection time increases. False negatives decrease for two reasons. First, the direct reduction; flows that take a long time to establish may take less than the new maximum connection time, resulting in a false negative turning into a successful result. The second, is less obvious; the increasing number of false positives from Bloom filter overloading result in some beneficial errors; flows that still exceed the maximum connection time, and therefore should be false negatives, are reported as being connected due to Bloom filter errors.

6.2.1 Computational Performance

To get a rough idea of the computational requirements and efficiency of SCD we ran our experiment program through the Linux profiler gprof (for trace C_04). Table 6.5 shows an abbreviated portion of the gprof output. The % time and self seconds columns represent the length of time spent in the function. The functions FindFlow, AgeFlows, RemoveFlow, and StoreFlow represent the computational requirements of our flow tracker, and the functions in bold represent SCD. To decrease the lookup time in the naive flow tracker, flow lookup was implemented using a one million element hash table, effectively reducing the number of lookup operations per packet to one, given that there are less than one million active flows. However even with this drastic attempt to increase the efficiency of the flow tracker the lookup time still accounts for 37% of execution time, and in total the naive flow

Table 6.5: Computational efficiency

%time	*selfsec*	*calls*	*name*
36.94	1105.5	161273588	FindFlow
18.75	561.12	322547176	StoreComps_o
14.19	424.76	330	AgeFlows
9.94	297.61	12225610	RemoveFlow
3.2	95.63	71668476	**HashLookup**
3.02	90.39	186368015	StoreFlow
1.06	31.85	41426043	**HashSet**
0.87	25.92	186368015	**SCD_Frame**
0.63	18.98	11798670	NewFlow
0.2	5.84	13808681	**HashAdd**
0.17	5.13	12322186	**PacketHash_rev**
0.15	4.53	11229407	**PacketHash_fwd**
0.11	3.39	14359349	**SCD_Packet**
0.09	2.72	12225610	ProcessFlowEnd

tracking functions account for 64%. By comparison, the performance advantage of an SCD implementation is clear. Only 5.76% of execution time, or 172.29 seconds (which is the sum of the SCD execution times, shown in bold) were required to process the 3600 second trace, leading to a 11x reduction in processing requirements.

In addition to the intrinsic efficiency of SCD, note that in typical network hardware many of the functions of SCD are implemented in hardware, such as the hash calculations, and bit-wise manipulation and addressing. If these functions were implemented in hardware it would essentially remove the PacketHash functions, and much of the processing time in HashLookup, HashSet, and HashAdd.

6.3 Experimental Performance of BDFT

In this section, we perform real-world experimental analysis of BDFT performance to verify the analytical results obtained in Chapter 5. Experimental analysis is performed using the

two Internet traffic traces described in section 6.1. To compare and validate the results from BDFT with the actual flow durations requires a 100% accurate flow tracker, similar to the one used in section 6.2, that is capable of reporting the real duration of every flow in the network. The duration of a flow is defined to be the time from the first SYN packet in the flow to the first FIN or RST packet. A tracking success is defined to be an estimated flow duration result that is within 50% of the actual flow duration, for flows that are greater than thirty seconds in duration. This definition is based on the fact that BDFT is intended to track long-duration flows to an approximate duration. In some situations a false negative can be returned by BDFT (and also by the comparison strategies, see section 6.3.1), these are always counted as a tracking failure.

The entire set of functionality required for BDFT was implemented in C. The bins were implemented using counting Bloom filters with the standard item insertion and removal policies. The basic BDFT functions of insertion, removal, searching, and aging the bins were implemented according to the descriptions of those functions given in Chapter 4. The design of the BDFT array is presented in section 5.4 for fine-grained tracking of flow duration. The basic bin sizing in Table 5.4 allocates 128k entries for the first bin, which we refer to as the base filter size. We tested BDFT at multiples of the base filter size (and therefore the same multiple of all the other filter sizes) to analyze the performance of BDFT at various memory usage levels.

As suggested in section 4.3.1, we include a SCD filter on the input to BDFT as well as a simple FIN/RST-checking Bloom filter. The SCD filter was sized large enough to be nearly 100% accurate to eliminate any affect on the BDFT performance results from SCD errors. The same SCD filter was also used on the input to the TDBF tracker (see section 6.3.1). Section 4.3.2 describes an issue that certain types of TCP timeouts cause for BDFT. Mitigating these effects is still an open area of research, so for our implementation we automatically remove flows after two minutes with no activity.

Finally, to perform a fair comparison of the performance of BDFT, we also implemented two other flow duration tracking schemes. The other schemes were configured with equal or greater available memory and sampling rate. Computational performance was not limited

or measured, so while other schemes may require more computational resources than BDFT, they were not penalized for increased CPU usage.

6.3.1 Other Flow Duration Strategies

The accuracy of BDFT, as presented in section 6.3.2 is a clear advantage. However, it is possible that another simpler implementation could achieve similar accuracy with less complexity. For this reason we selected two other strategies to track flow duration and compared the results with BDFT. In this section we describe the operation of the two competing duration tracking strategies.

To maintain similarity in the comparisons the memory usage and sampling rate were balanced between all three strategies, BDFT, Naive Sampled, and TDBF.

Naive Sampled

A simple approach to tracking flow duration is based on the sampling idea used by NetFlow. NetFlow operates by keeping a flow record for every flow that it sees in the network. When a packet arrives its corresponding flow record is looked up and the appropriate statistics and counters updated. If no flow record exists for the flow then a new one is created. Flow records are removed if no packets are received from that flow for a short period of time, typically 15 seconds. Likewise, flows records are also expired once they reach a certain age, typically 15-30 minutes. When a flow record expires it is exported to an external collector computer for further processing.

NetFlow has a number of disadvantages and problems when scaling to 10Gbps routers which make it unsuitable for directly tracking flow duration. The first issue that arises is that the timeouts required by NetFlow mean that any flow that is idle will timeout and its duration will be lost, and more critically the maximum flow duration that can be tracked is 15-30 minutes. NetFlow also has additional issues when scalability is considered. The memory required to store the flow records quickly exceeds the capabilities of SRAM, and due to the requirement to perform a search and memory writes with every packet received NetFlow can not be implemented using DRAM when tracking every flow in a network. The

solution to the scalability problems of NetFlow is to perform packet sampling where only one in every hundred or thousand packets is sent to NetFlow for processing. Sampling raises further problems when attempting to track flow duration, as discussed further in section 6.3.2.

The pure naive idea to track flow durations is to process every packet and keep a record of every flow in the network. As part of each flow record timestamps could be maintained for the first and last packet in the flow, and the flow records could be expired after a long interval. Unfortunately this approach would share the same scaling problems as basic NetFlow. So in our implementation of a naive duration tracker we make several sacrifices to the overall accuracy to obtain time and space performance that is comparable to BDFT.

For our naive duration tracker implementation the tracker processes packets based on a sampling rate of one in twenty, which matches BDFT's effective sampling rate as explained in section 4.3.4. The sampled packets are processed in a similar way to NetFlow. A flow record table is maintained which contains entries consisting of a flow identifier and the start and last packet timestamps. When a packet is sampled the table is searched for the flow and the timestamps updated if the flow is already in the table, or the flow is added to the table. If the table is full then the oldest flow in the table is expired, meaning it is removed from the table.

Accuracy results for the naive implementation are determined by searching the flow table every time a flow terminates. The estimated duration for the flow returned from the naive tracker is compared with the actual time using the same comparison methods used for BDFT.

Time-Decaying Bloom Filters

The second approach to tracking the duration of flows involves the use of a single Time-Decaying Bloom filter. A time-decaying Bloom filter allows the multiplicity of an item in a set to be tracked, with more recent insertions being weighted higher. Section 2.3 explains the basic operation of time-decaying Bloom filters. We can modify the basic operation of the TDBF slightly to track the length of time that an item has been in a set. The basic

idea is to initialize all counters corresponding to an item's hashes to a large initial value, e.g., 40000. Then, in our implementation, every ten seconds all counters are decremented. When a query is performed to determine how long the item has been in the set the duration is equal to (40000 - current counter value) * 10 seconds. The detailed operation of the modified TDBF can be explained through an explanation of its main operations, insert, search, and removal.

To insert a flow into the TDBF all counters corresponding to the items hashes are set to their maximum value according to how many bits are available in a counter. For our implementation we used 16 bit counters for a maximum value of 65535. Counters are set to the maximum value regardless of their current value.

To remove a flow from the TDBF all of the corresponding counters are set to 0.

To determine the duration of a flow the counters corresponding to the flows hashes are read and the lowest value that is not 0 is selected. The duration can be calculated by the formula (65535 - lowest counter val) * 10 seconds. This formula is based on the fact that we decrement all counters every 10 seconds. If all counters are 0 then a failure to find the flow is reported. It is important to note that choosing the lowest counter that is not 0 breaks the fundamental operation of Bloom filters where a 0 would normally indicate that the item is not in the filter. Due to the nature of network traffic and assuming that queries will only be performed for flow IDs that are guaranteed to be in the filer we can assume that if any counters are not 0 then the flow has not been removed. This optimization in our specific case results in a 100% improvement in accuracy. If the above assumption can not be met then flows which have one or more counters set to 0 can not be assumed to be in the network and must be returned as a lookup failure.

In terms of complexity when compared to BDFT the time-decaying Bloom filter is somewhat simpler due to the fact that there is only a single Bloom filter instead of one per bin.

Table 6.6: Algorithm performance comparison for *C_04*

Algorithm	Memory Usage (bytes)	Number of Overflowed Counters	Accuracy
BDFT	90112	902	79.24%
BDFT-FPC	90112	974	86.50%
BDFT	180224	134	96.15%
BDFT-FPC	180224	158	97.13%
BDFT	360448	16	99.59%
BDFT-FPC	360448	18	99.84%
BDFT	720896	1	99.98%
BDFT-FPC	720896	1	99.98%
Naive (1in 20)	524288	n/a	6.07%
Naive (1in 100)	524288	n/a	1.58%
TDBF	131072	n/a	58.79%

6.3.2 BDFT Performance Results

Tables 6.6 and 6.7 show the performance of BDFT and BDFT-FPC (section 4.3.3) with various memory configurations. These tables also show the TDBF and Naive results for comparison. The BDFT accuracy results exceeded our expectations. With only 0.257 bits of storage required per flow BDFT achieves 99.59% accuracy on the *C_04* trace, and with only 0.128 bits of storage required per flow BDFT achieves 99.55% accuracy on the *N_12* trace. Bits per flow is calculated by dividing the memory usage by the number of established flows in an hour (352,410 for *N_12* and 11,215,873 for *C_04*), as shown in Table 6.3. The number of overflowed counters column is higher than the theoretical expectations derived in section 5.4. This is due to real-world factors not taken into account during the theoretical analysis, such as timed-out flows, and errors introduced by the SCD filtering, and is discussed further below.

The significant difference in per-flow memory requirements for the two traces is caused by the differences in the flow duration distribution for the two flows. Referring back to Figures

Table 6.7: Algorithm performance comparison for *N_12*

Algorithm	Memory Usage (bytes)	Number of Overflowed Counters	Accuracy
BDFT	1408	912	38.43%
BDFT-FPC	1408	920	52.81%
BDFT	2816	136	82.23%
BDFT-FPC	2816	137	93.33%
BDFT	5632	17	99.55%
BDFT-FPC	5632	17	99.58%
BDFT	11264	1	99.97%
BDFT-FPC	11264	1	99.97%
BDFT	22528	0	100.00%
BDFT-FPC	22528	0	100.00%
Naive (1in 20)	524288	n/a	8.34%
Naive (1in 100)	524288	n/a	3.82%
TDBF	131072	n/a	90.84%

6.1 and 6.2, we observe that the distribution for the *N_12* trace is tightly concentrated in the 0-4 second range, whereas C_04 is relatively evenly distributed to higher durations. When the flow durations are concentrated such that most flows end within one or two seconds, the overall loading on the first and therefore subsequent bins is greatly reduced. In this situation flows are added to the first bin and then quickly removed so the number of flows actually stored in the bin at any time is quite low, resulting in lower memory requirements.

Tables 6.6 and 6.7 illustrate two important additional points. BDFT is able to achieve 100% (or near 100%) accuracy when the available memory is increased to the point where almost all counter overflows are eliminated. Therefore, even though BDFT is an approximate method of tracking flow duration, it is applicable in situations where close to 100% accuracy is required if sufficient memory is available. The memory requirements for achieving 100% or close to 100% accuracy were 10.37 bits/flow for the *C_04* trace, and 0.657 bits/flow for the *N_12* trace.

The second performance related result addresses the performance of BDFT under heavy

load. By reducing the available memory to far below the expected requirements we were able to simulate overload conditions for BDFT. For the *C_04* this was at 90,112 bytes and for *N_12* it was at 1408 bytes. At these memory levels the number of overflowed counters and false positive rates are far above normal design limits, resulting in reduced accuracy. However BDFT continues to behave in a consistent manner, returning correct results for many flows, instead of suffering a catastrophic breakdown of accuracy. This behavior is important for deployment in real network conditions where spikes in traffic could temporarily overload a BDFT array.

We observed several interesting results from the BDFT performance testing. The number of overflowed counters is substantially higher than expected. We theorize that this is due to the number of flows that end up in the timeout state, as defined in section 4.3.2. The flows which timeout end up "polluting" the higher duration bins with many dead flows. In our case these bins were not designed to handle the increased number of flows due to the timeouts, and therefore can become overloaded to the point of having overflowed counters. This problem can be easily corrected by increasing the counter width to three bits. Table 6.3 shows the number of timed out flows for each trace.

Figures 6.6 and 6.7 compare the accuracy results for BDFT and BDFT-FPC for the two traces. In both cases BDFT-FPC provides an increasing benefit as the accuracy decreases. This effect is due to fact that the number of false positive removals (see section 4.3.3) will increase as the probability of a false positive increases as the bins become increasingly full. As the number of entries in the filters increases the number of overflowed counters will also increase, this is the second major contribution to the number of errors reported by BDFT. The benefits of implementing BDFT-FPC are high when the majority of errors are caused by false positive removals, however if most errors are caused by overflowed counters then BDFT-FPC will result in only small gains in accuracy. The cause of error in BDFT, false positives or overflows, will vary based on the array configuration, e.g. 4-bit wide counters will almost eliminate counter overflows. Therefore each BDFT array configuration must be evaluated individually to determine the benefits of adding FPC.

In Figure 6.7, the BDFT and BDFT-FPC lines are almost parallel when the memory

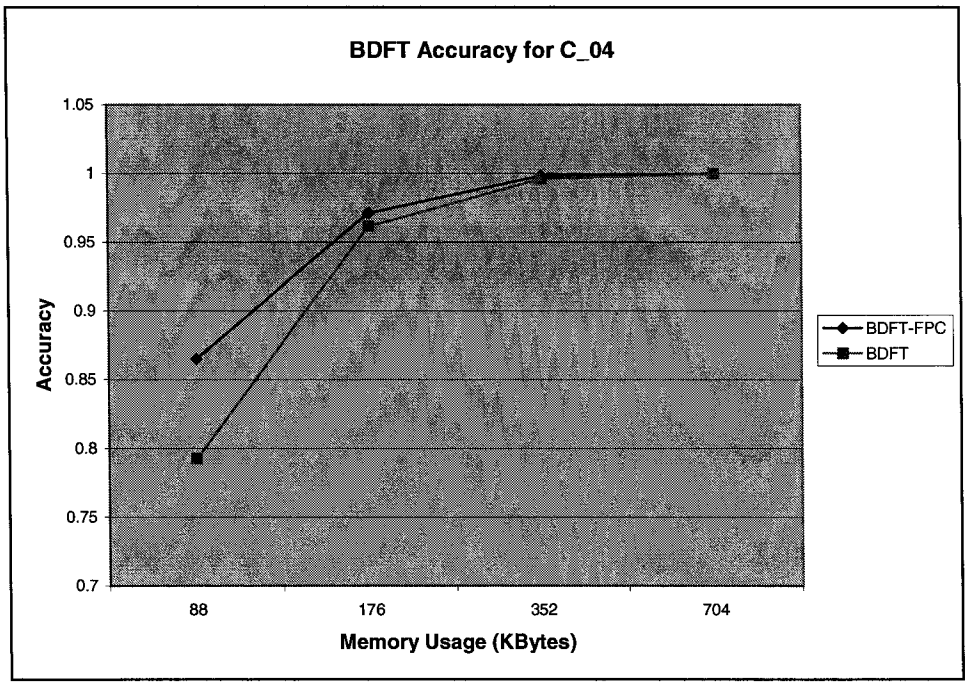

Figure 6.6: Accuracy of BDFT and BDFT-FPC for *C_04*

increases from 1408 to 2816 bytes. This indicates that at these memory levels the number of BDFT errors is dominated by the number of overflowed counters, and false positive removals have only a small effect. In comparison, in Figure 6.6 the benefit of BDFT-FPC can be clearly seen between the 88K and 176K memory levels. From a point where BDFT and BDFT-FPC are almost equal at 176K bytes, at 88K bytes BDFT-FPC is about 7% more accurate than BDFT, which is a result of the false positive correction.

Figures 6.8 and 6.9 are the first of a series of scatter plots which demonstrate the accuracy of BDFT and the other algorithms in a visual way. Flows are plotted as a point where the actual flow duration and the flow duration estimated by BDFT (or Naive/TDBF) meet. Flows that land on the the arrow through the diagonal of the graph are perfectly accurate, and therefore any deviation from the centerline indicates some inaccuracy in the duration reported. Flows which are false negatives are shown as an estimated duration of one, along the bottom of the graph.

Figure 6.8 shows the best BDFT performance for the *C_04* trace. The base filter size

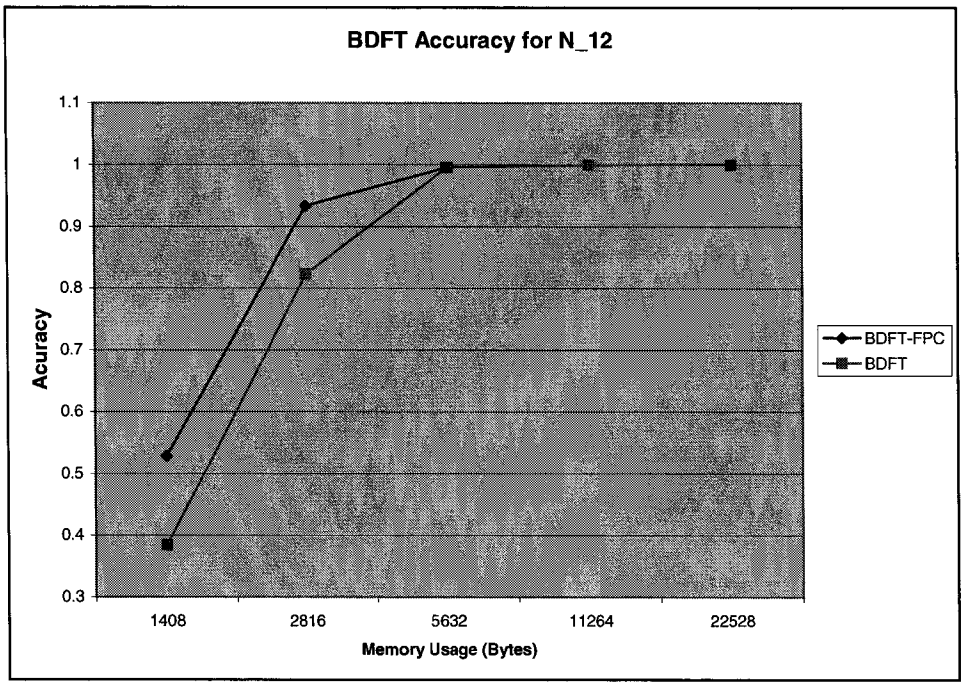

Figure 6.7: Accuracy of BDFT and BDFT-FPC for *N_12*

was 512k entries and FPC was on, resulting in 99.98% accuracy. Figure 6.9 shows a BDFT implementation where reduced memory and computational requirements must be met. A base filter size of 128k was used resulting in 180Kbytes of memory usage and 96.15% accuracy. The plot shows that some flows are false negatives (along the bottom of the graph), and quite a few short duration flows are false positives in higher duration bins.

Figures 6.10 and 6.11 show the BDFT results for the *N_12* trace. Two memory and FPC configurations were selected which represent design requirements to have; 100% accuracy, and low memory and computational requirements, respectively. Figure 6.10 shows the results for a 100% accurate BDFT configuration of a base filter size of 16k and FPC turned on. Figure 6.11 shows 99.55% accuracy with a base filter size of 4k and FPC off. In this figure it is clear that the highest duration bins became overloaded and were reporting many false positives for short duration flows in the 2205-3165 second and 3165-3600 second bins.

Figures 6.12 and 6.13 show Naive (1 in 20 sampling) and TDBF results for the *C_04* trace. The Naive implementation is only 6.07% accurate in for this trace, with most flows

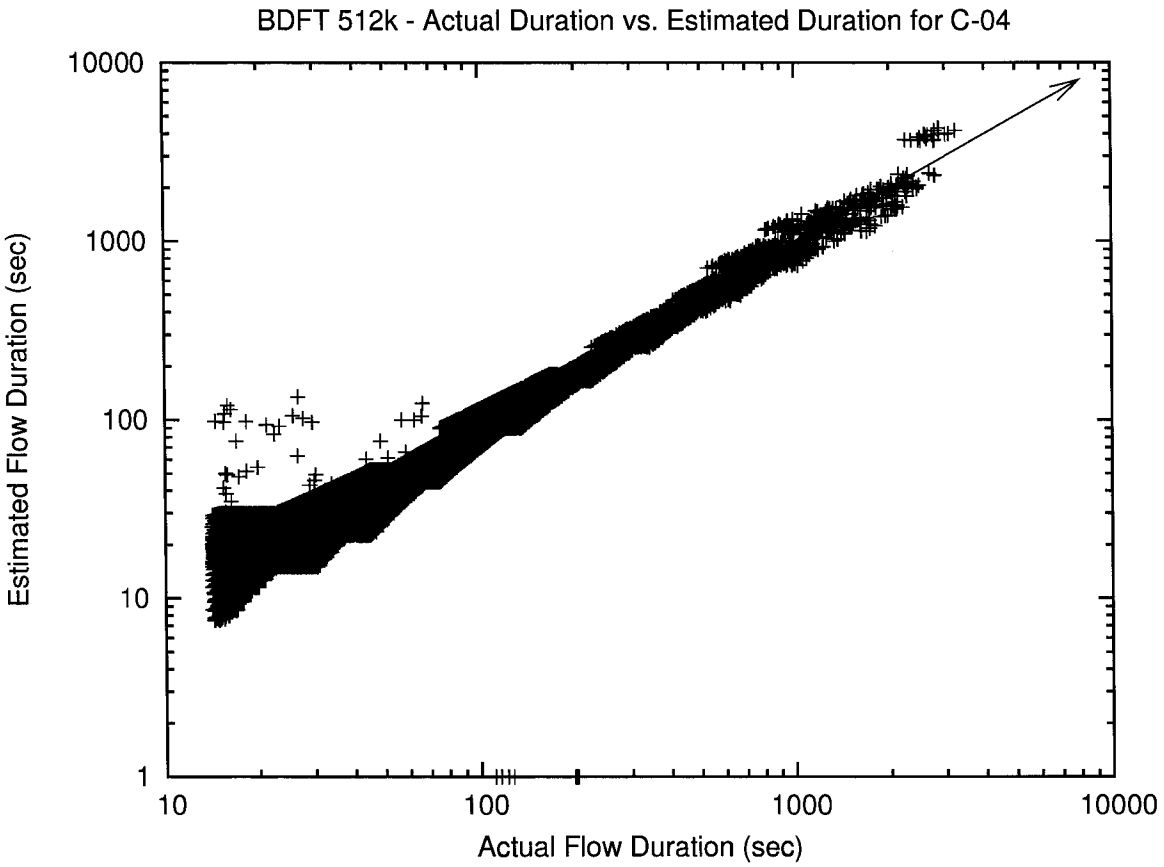

Figure 6.8: BDFT performance 512k FPC - *C_04*

being reported as "not tracked" and ending up along the bottom of the graph. As expected the naive implementation always under-reports the duration of flows, sometimes severely. Of note is that the naive implementation does manage to track some long duration flows that are also high bandwidth flows, due to the fact that high bandwidth means the probability of them being sampled is high. The TDBF approach is 58.79% accurate and both under-reports and over-reports the duration of flows. The notable point for TDBF performance is the fact that no flow durations over 1000 seconds are reported by TDBF and there are very few over even 500 seconds. This means that the longest duration flows, which are potentially the ones the operator is most interested in, are not tracked successfully.

Figures 6.14 and 6.15 show Naive (1 in 20 sampling) and TDBF results for the *N_12* trace. The Naive implementation is only 8.34% accurate in for this trace, and therefore,

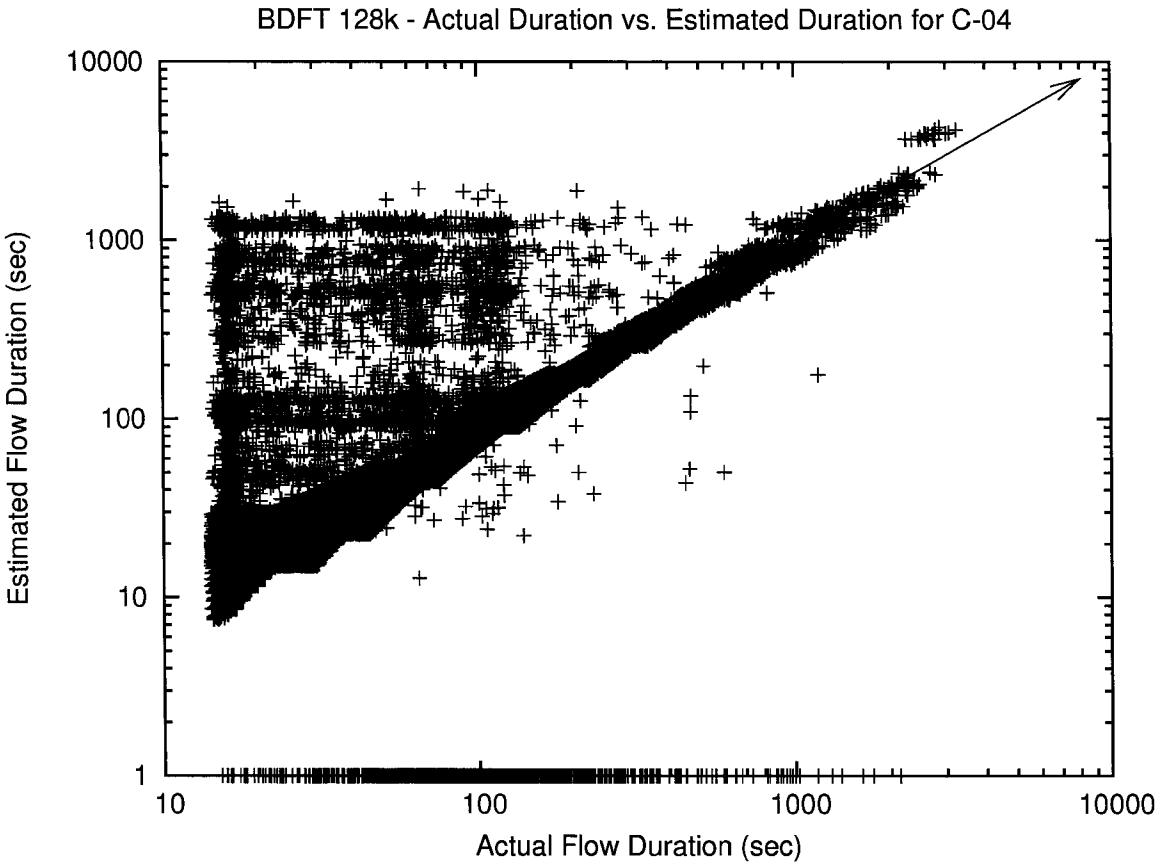

Figure 6.9: BDFT performance 128k - *C_04*

like with the *C_04* trace, most flows are reported as "not tracked" and end up along the bottom of the graph. Also, similar to the *C_04* trace some long-duration high-bandwidth flows are tracked successfully. TDBF was quite accurate for this trace, at 90.84%, due to the relatively large amount of memory allocated for it (about 47x the BDFT requirements for the same accuracy). However, TDBF has trouble tracking flows over 1000 seconds long, these flows are either mis-reported or TDBF fails to report any value for the flow.

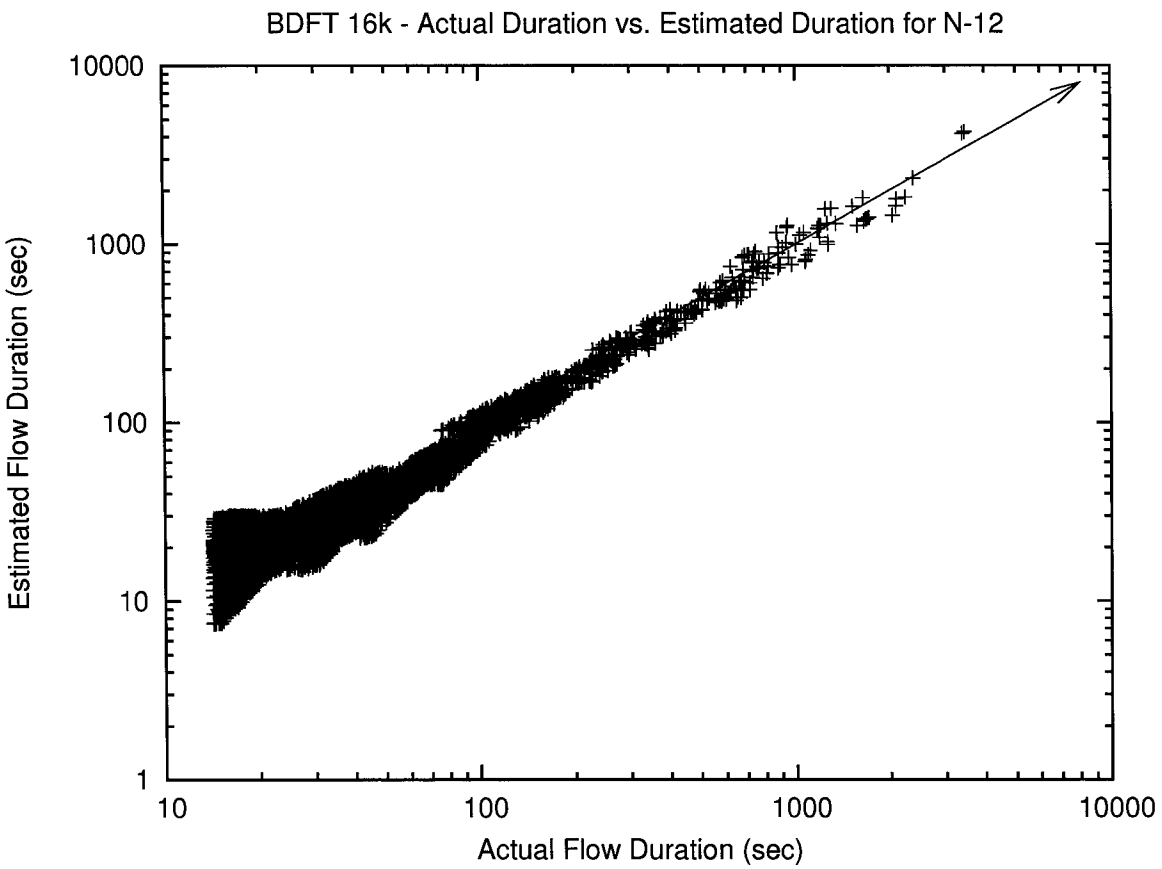

Figure 6.10: BDFT performance 16k FPC - *N_12*

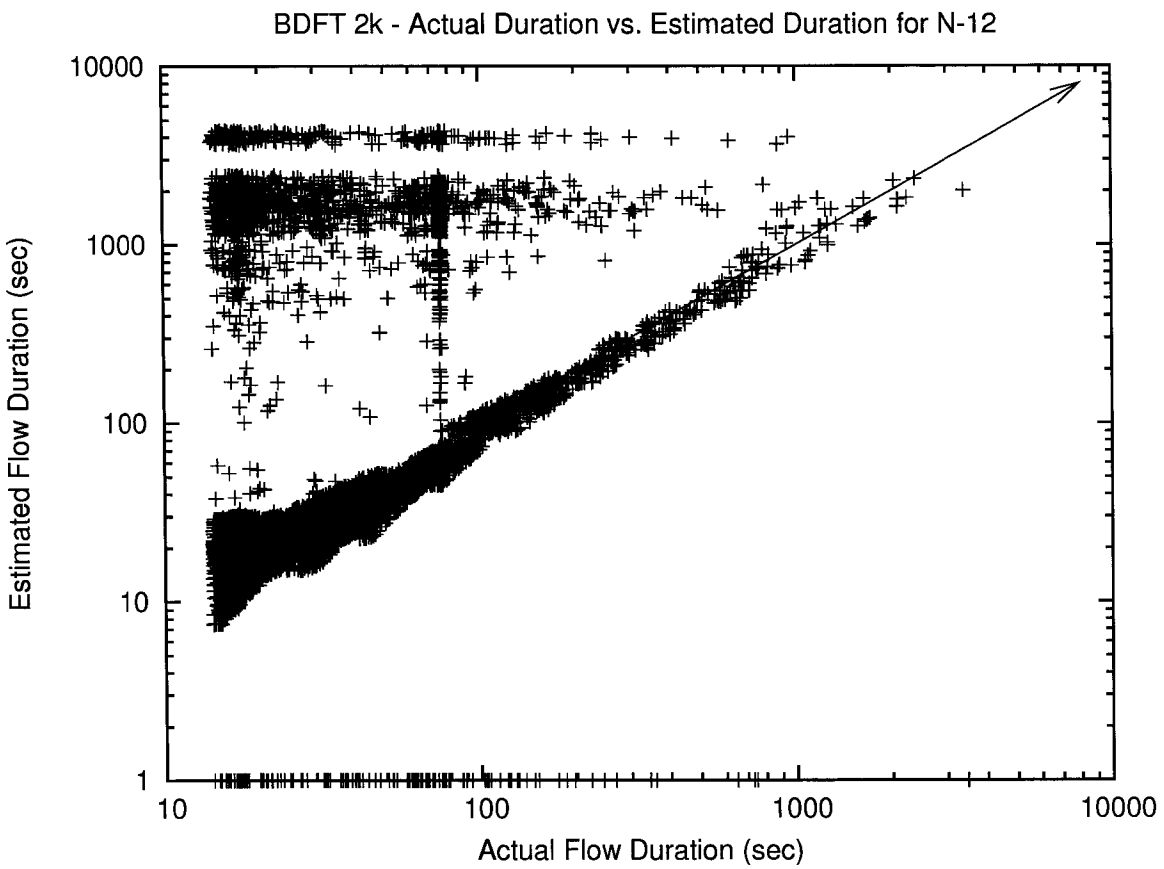

Figure 6.11: BDFT performance 2k - *N_12*

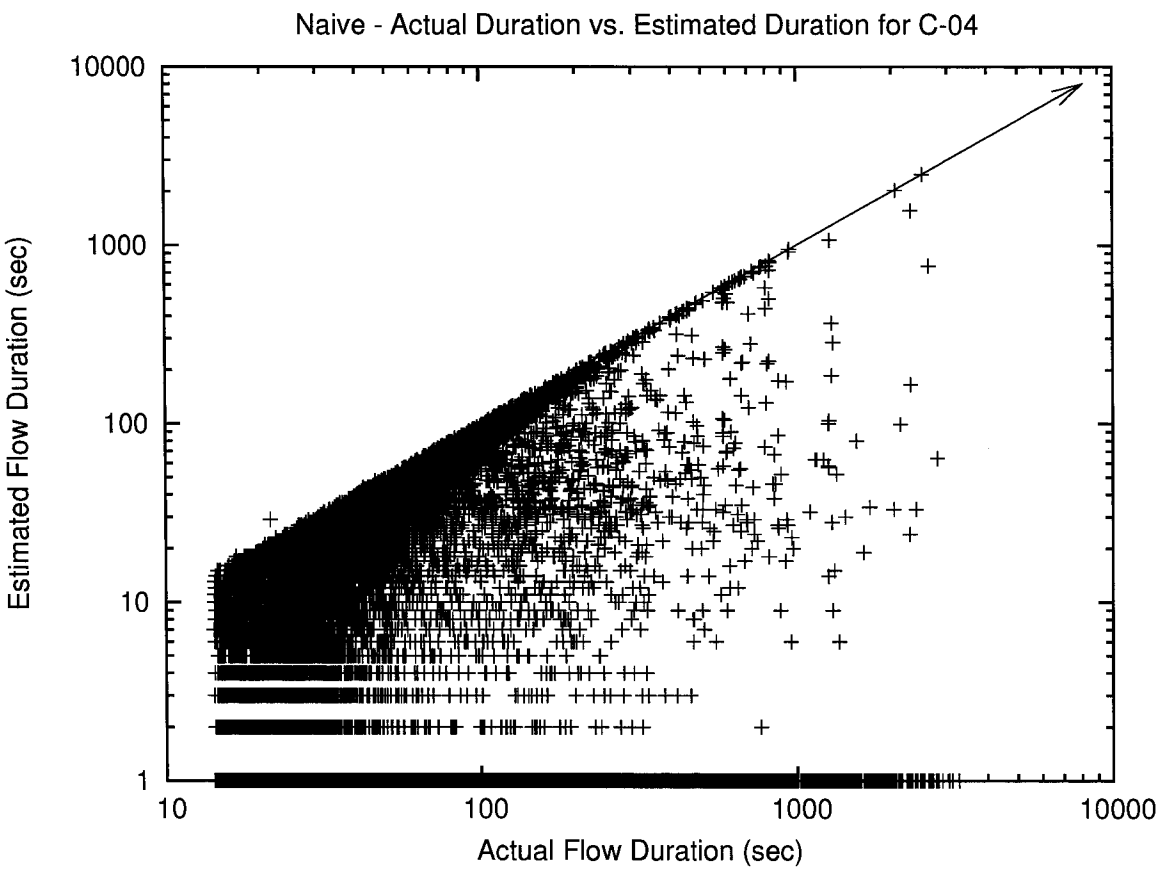

Figure 6.12: Naive performance (1 in 20) - *C_04*

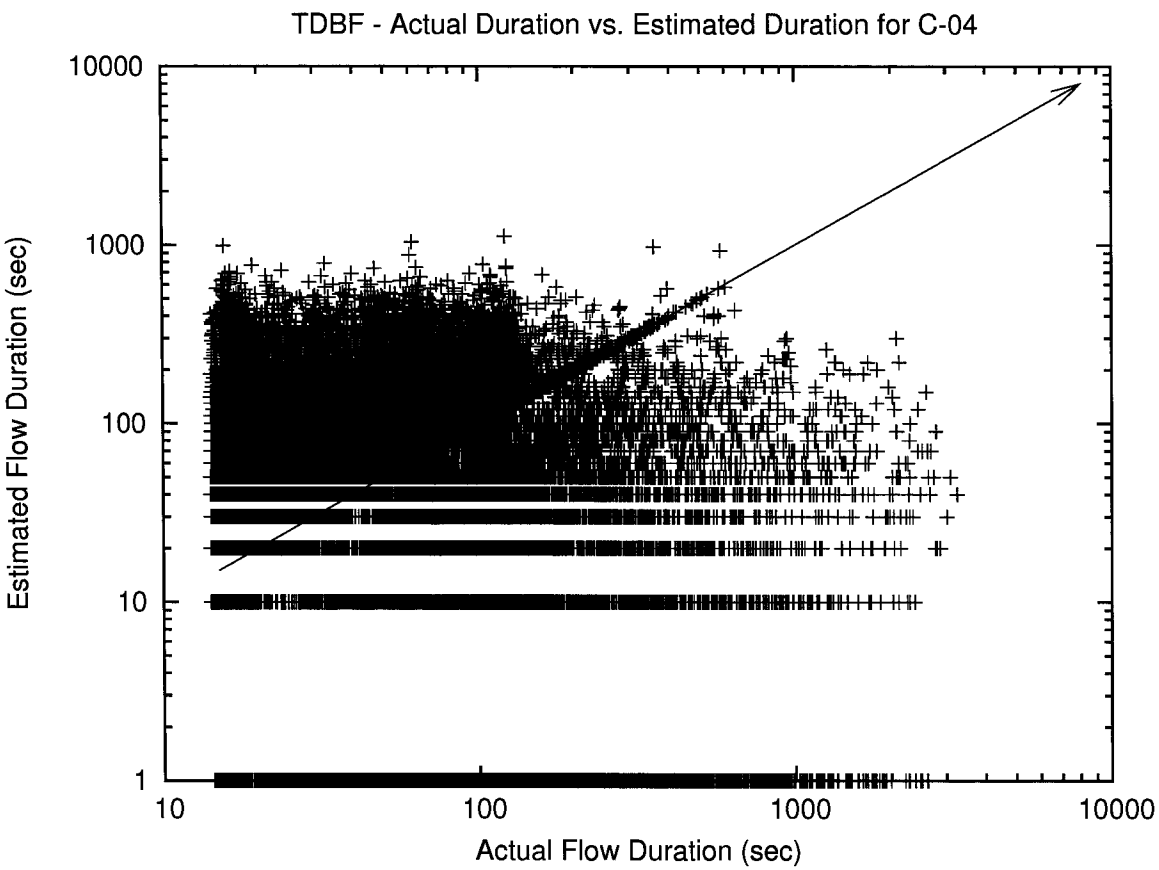

Figure 6.13: TDBF performance - *C_04*

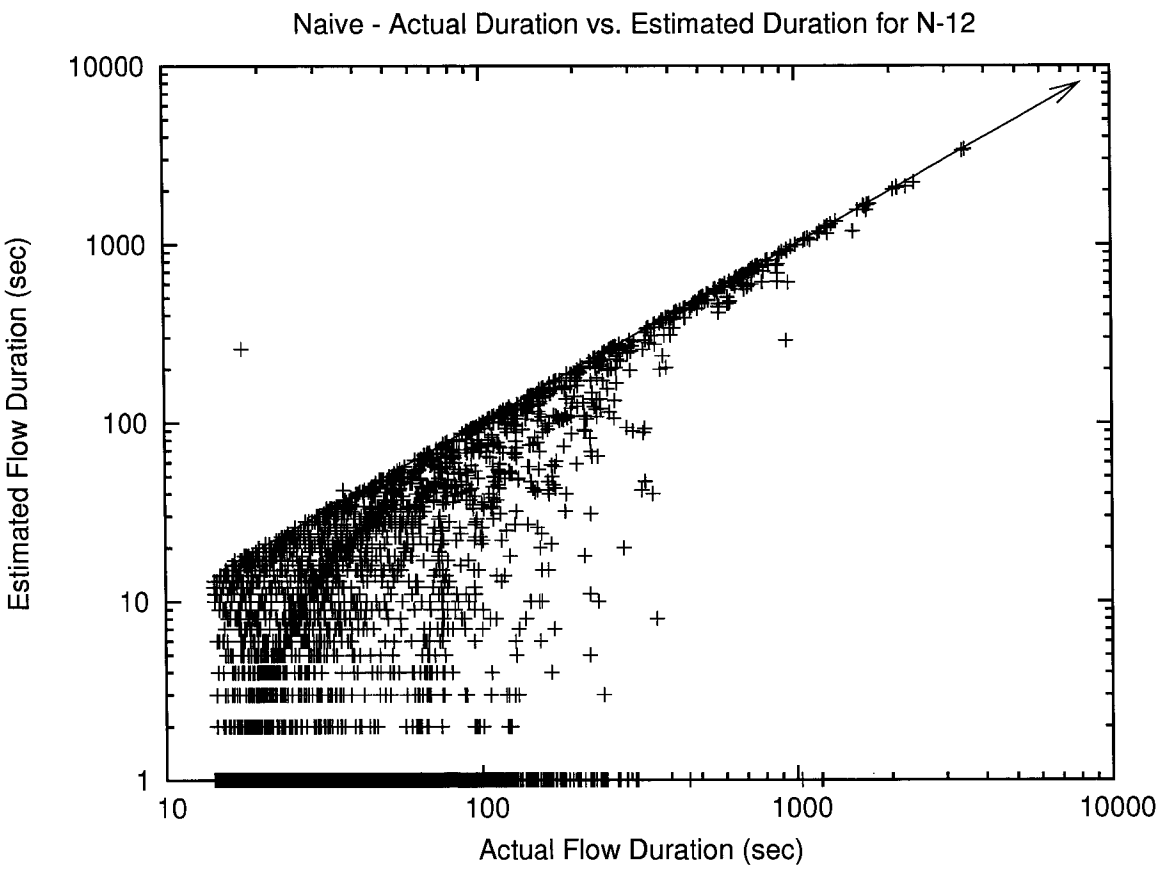

Figure 6.14: Naive performance (1 in 20) - *N_12*

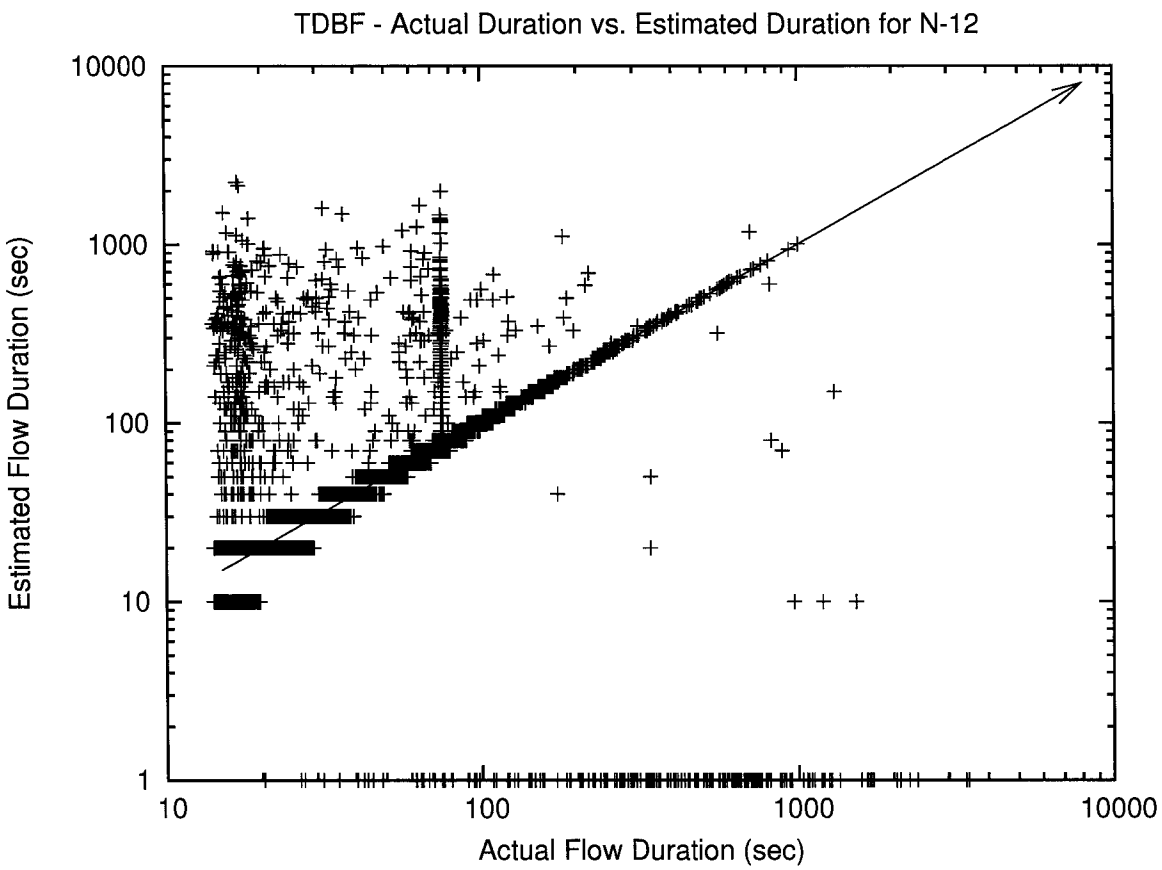

Figure 6.15: TDBF performance - *N_12*

Chapter 7

Concluding Remarks

This thesis presented two complimentary methods of performing network monitoring on a per-flow basis, and for every flow observed on a network device. The first method, Symmetric Connection Detection (SCD), filters incomplete flows out of a network stream, passing only fully established flows on to a secondary flow tracking application. The second method, Binned Duration Flow Tracking (BDFT), is a per-flow method of tracking the approximate duration of all flows in a network, on a per-flow basis. Both methods introduce novel concepts to the design of algorithms intended for deployment on high-speed routers. Before our work on flow duration there was no acceptably accurate way to track flow duration on high-speed routers, to the best of our knowledge.

We have shown that SCD provides a viable real-time method of reporting fully established TCP flows. Using very little memory, SCD is able to achieve accuracy of 99%+. In addition, SCD can be implemented using hardware-based bloom filters or on network processors that use SRAM memory. The parameters of SCD are flexible and only need to be set to approximately the ideal value to achieve high accuracy.

BDFT was shown to approach 100% accuracy with low per-flow memory usage, and is able to classify flows to their correct state and duration. The design of a BDFT array is flexible and can be customized to the needs of the network operator, for example, durations can be tracked on a fine-grained basis, or BDFT can be used to classify flows as short, medium, and long. We also showed that BDFT is suitable for deployment on high-speed routers when coupled with SCD, and detailed the expected theoretical and real-world accuracy and

computational requirements of BDFT.

The successful coupling of SCD and BDFT high-lights the effectiveness of SCD when used to reduce the computational requirements placed on secondary processing applications. We have demonstrated that SCD can reduce the number of flows that must be processed by 95%, depending on the traffic mix. We anticipate that many network flow tracking applications desire to process only those flows that complete their connection process, and therefore will greatly benefit from the pre-filtering that SCD provides.

BDFT opens up a potential research area that was previously not available - the classification of flows according to duration on high-speed routers. Duration classification opens a number of new potential research areas. Duration-based routing (making routing and packet scheduling decisions based on flow duration) could have a major impact on the quality of service of general Internet traffic. BDFT will allow packet classification to rely on real-time flow duration data, making real-time transport level classification, such as Peer-To-Peer classification, feasible. When flow duration is combined with other traffic classification metrics it may be possible to make automatic packet scheduling decisions based on application type. As a final note, external network traffic analysis applications now have a way, through BDFT, to access flow duration data in real-time, which could increase the accuracy of analysis applications.

As an area of future work it may be possible that SCD can be used for port scan detection and detection of some attacks, with slight modifications. The bloom filters can be modified to counting bloom filters, and the hashes can be based on IP addresses only. In this way it would be possible to track the number of failed connection attempts on a per-IP basis, with some errors.

BDFT has several areas of potentially valuable future work. First off, the generalization of BDFT to tracking state instead of duration requires further investigation. The techniques presented in the "Beyond Bloom Filters" paper could be further evaluated for their applicability to tracking duration, and a state-generalized BDFT performance could be evaluated against the methods presented in the paper. As mentioned in section 4.3.2 an efficient means of handling certain flow timeout conditions is required and would increase

the long-run accuracy of BDFT, this issue is also mentioned in "Beyond Bloom Filters".

In conclusion, I would once again like to thank those people and organizations mentioned in the Acknowledgments section. This research would not be possible without the resources and support generously provided.

List of References

[1] J. Quittek, T. Zseby, B. Claise, and S. Zander, "Netflow version 9,." Cisco Systems `http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html` (last accessed on Dec 16, 2006).

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[3] K. Shah, S. Bohacek, and A. Broido, "Feasibility of detecting TCP-SYN scanning at a backbone router," in *Proceedings of the American Control Conference*, pp. 988–995, Jul 2004.

[4] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy, "Transport layer identification of P2P traffic," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, pp. 121–134, Oct 2004.

[5] S. Sen, O. Spatscheck, and D. Wang, "Accurate, scalable in-network identification of P2P traffic using application signatures," in *Proceedings of the 13th International Conference on World Wide Web*, pp. 512–521, 2004.

[6] K. Tutschku, "A measurement-based traffic profile of the edonkey filesharing service," in *Proceedings of the 5th Passive and Active Measurement Workshop*, Apr 2004.

[7] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy, "An analysis of internet content delivery systems," *SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 315–327, 2002.

[8] D. Erman, D. Ilie, and A. Popescu, "Bittorrent traffic characteristics," in *Proceedings of the International Multi-Conference on Computing in the Global Information Technology*, pp. 42–56, 2006.

[9] J. Erman, M. Arlitt, and A. Mahanti, "Traffic classification using clustering algorithms," in *Proceedings of the 2006 SIGCOMM Workshop on Mining Network data*, pp. 281–286, 2006.

[10] A. Z. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proceedings of INFOCOM*, pp. 1454–1463, Apr 2001.

[11] C. Estan and G. Varghese, "Data streaming in computer networking," in *Workshop on Management and Processing of Data Streams*, 2003.

[12] A. Kumar, J. J. Xu, L. Li, and J. Wang, "Space-code Bloom filter for efficient traffic flow measurement," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet measurement*, pp. 167–172, 2003.

[13] P. Phaal, S. Panchen, and N. McKee, "Inmon corporation's sflow: a method for monitoring traffic in switched and routed networks,." `http://www.inmon.com/technology/index.php` (last accessed on Dec 16, 2006).

[14] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, pp. 153–166, 2003.

[15] J. Networks, "Enabling source class and destination class usage,." `http://www.juniper.net/techpubs/software/junos/junos70/swconfig70-interfaces/html/interfaces-family-config25.html` (last accessed on Dec 17, 2006).

[16] N. Brownlee, C. Mills, and G.Ruth, "Traffic flow measurement: architecture (RFC 2722),." `http://www.faqs.org/rfcs/rfc2722.html` (last accessed on Dec 17, 2006).

[17] L. Degioanni and G. Varenni, "Introducing scalability in network measurement: toward 10 gbps with commodity hardware," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, pp. 233–238, 2004.

[18] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 323–336, 2002.

[19] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," *SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 245–256, 2004.

[20] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 325–336, 2003.

[21] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: a survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[22] M. E. Attig and J. Lockwood, "SIFT: Snort intrusion filter for TCP," in *Symposium on High Performance Interconnects*, pp. 121–127, Aug 2005.

[23] T. Kocak and I. Kaya, "Low-power Bloom filter architecture for deep packet inspection," *Communications Letters, IEEE*, vol. 10, pp. 210–212, Mar. 2006.

[24] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area Web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[25] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Computer and System Sciences*, vol. 7, no. 4, pp. 448–461, 1973.

[26] S. Muthukrishnan, "Data streams: algorithms and applications," 2003. `http://www.cs.rutgers.edu/~muthu/` (last accessed on Dec 16, 2006).

[27] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Space- and time-efficient deterministic algorithms for biased quantiles over data streams," in *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 263–272, 2006.

[28] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss, "Domain-driven data synopses for dynamic quantiles," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 7, pp. 927–938, 2005.

[29] D. Reed, "Ipfilter." `http://coombs.anu.edu.au/~avalon/` (last accessed on Dec 16, 2006).

[30] "Snort." `http://www.snort.org/` (last accessed on Dec 16, 2006).

[31] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[32] N. Weaver, S. Staniford, and V. Paxson, "Very fast containment of scanning worms," in *Proceedings of the 13th USENIX Security Symposium*, pp. 29–44, Aug 2004.

[33] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 270–313, 2003.

[34] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom filters: from approximate membership checks to approximate state machines," *SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 315–326, 2006.

[35] D. I. program, "Transmission control protocol,." `http://www.faqs.org/rfcs/rfc793.html` (last accessed on Dec 17, 2006).

[36] N. Brownlee and K. Claffy, "Understanding internet traffic streams: dragonflies and tortoises," *IEEE Communications Magazine*, vol. 40, pp. 110–117, Oct 2002.

[37] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables," in *Proceedings of the 15th ACM-SIAM symposium on Discrete algorithms*, pp. 30–39, 2004.

[38] C. Shannon, E. Aben, kc claffy, D. Andersen, and N. Brownlee, "The CAIDA OC-48 traces dataset, collected in April, 2003,." `http://www.caida.org/data/passive/` (last accessed on Dec 17, 2006).

[39] N. L. for Applied Network Research, "NCAR-1 trace, collected in December, 2003, NSF ANI-0129677 (2002) and ANI-9807479 (1998),." `http://pma.nlanr.net/Special/ncar1.html/` (last accessed on Dec 17, 2006).

Appendix A

IP Hash Functions

These are the IP and TCP port hash functions used to generate the three Bloom filter hashes for experimental testing of BDFT and SCD.

```
void PacketHash_ip( unsigned int daddr,
                                    unsigned int saddr,
                                    unsigned short source,
                                    unsigned short dest,
                                    unsigned int *hash1,
                                    unsigned int *hash2,
                                    unsigned int *hash3 )
{
        unsigned int    hash;

        hash = 0x9E34B213;
        hash += daddr + dest;
        hash ^= hash << 7 ^ hash >> 17;
        hash += (saddr * 7) + source;
        hash ^= hash << 9 ^ hash >> 5;
        hash += ((unsigned int) source) * 13 ^ (dest << 9);
        hash ^= hash >> 3;
```

```
        hash += ((unsigned int) dest) * 37 ^ (source << 17);
        hash ^= hash << 5;
        *hash1 = hash;

        hash = 0x5B39AF34;
        hash += (daddr * 5) + saddr;
        hash ^= hash << 9 ^ hash >> 5;
        hash ^= (saddr * 13) + source + dest ;
        hash ^= hash << 11 ^ hash >> 3;
        hash += ((unsigned int) source) * 3 ^ (dest >> 1);
        hash ^= hash << 3;
        hash += ((unsigned int) dest) * 127 ^ (source << 11);
        hash ^= hash << 7 ^ hash >> 11;
        *hash2 = hash;

        hash = 0xB599CD98;
        hash ^= ((daddr << 4) ^ (daddr >> 7)) + dest ;
        hash ^= hash << 15 ^ hash >> 7;
        hash ^= ((saddr << 5) ^ (saddr >> 3)) + source;
        hash ^= hash << 3 ^ hash >> 11;
        hash ^= (((unsigned int) source) << 13) ^
                (((unsigned int) source) >> 3) + (source * 11);
        hash ^= hash >> 7;
        hash ^= (((unsigned int) dest)<< 17) ^
                ((unsigned int) dest) + (dest * 17);
        hash ^= hash << 9;
        *hash3 = hash;
}
```