2008

Effective techniques for detecting and attributing cyber criminals

Linfeng Zhang
Iowa State University

Effective techniques for detecting and attributing cyber criminals

by

Linfeng Zhang

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Yong Guan, Major Professor
Thomas E Daniels
Julie A Dickerson
Douglas W Jacobson
Arun K Somani
Johnny S Wong

Iowa State University

Ames, Iowa

2008

DEDICATION

I would like to dedicate this thesis to my wife Danhui without whose support I would not have been able to complete this work. I would also like to thank my mother Hui, my father Chongtao, my father-in-law Renjie, my mother-in-law Shaohua Chen, my brothers Xuefeng, Yunfeng and their families, for their endless love.

And, to my lovely son, Ryan.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, I am deeply grateful to my advisor, Dr. Yong Guan. Without his knowledge, perceptiveness, patience, guidance and support, I would have never finished my thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. During the long rejection-rejection-····-acceptance circles of my publications, he always encouraged me, and gave me enough confidence to not give up. I would remember the moments when we bore the failures together and finally shared the successes for ever. I would also remember his advices not only to my research but also to my life and career.

I would like to thank Dr. Thomas Daniels and Dr. Julie A Dickerson for their invaluable guidance and advices when I was working on the Advanced Attack Attribution project. I would also like to thank my other committee members for their efforts and contributions to this work: Dr. Douglas W Jacobson, Dr. Arun K Somani and Dr. Johnny S Wong. Their suggestions inspired me and improved the quality of my thesis. I would additionally like to thank all the lecturers in Department of Electrical and Computer Engineering and other departments, from whom I learnt a lot. Also, many thanks to Mrs. Pamela J Myers and other department staffs for their help in last five years.

I would like to thank all my labmates and friends at Iowa State University. I am only able to list a few here: Su Chang, Bryan Ellingson, Alan Johnson, Yang Liu, Wale Martins, Yanlin Peng, Anthony G. Persaud, Yongping Tang, Wei Wang, Yawen Wei, Jianqiang Xin and Zhen Yu.

I appreciate that all of you provided me a wonderful life in Ames, Iowa between August

2003 and December 2008. I would remember your faces, your words, and your deeds, deeply and gratefully.

This work was partially supported by NSF under grants No. CNS-0644238, CNS-0626822, and DUE-0313837, ARDA under contract number NBCHC030107, and Carver Trust Foundation.

ABSTRACT

With the phenomenal growth of the Internet, more and more people enjoy and depend on the convenience of its provided services. Unfortunately, the number of network-based attacks is also increasing very quickly. More and more fraud activities appear in online advertising networks and online auction systems. Network attackers can easily hide their identities through IP spoofing, stepping stones, network address translators, Mobile IP or other ways, and thereby reduce the chance of being captured. The current IP network infrastructure lacks measures and cannot effectively deter and identify motivated and well-equipped attackers. Therefore, innovative traceback schemes are required to attribute the real attackers. By the way, network traffic always comes with high rate in distributed format without obvious beginning and ending. These properties make network traffic much different compared with traditional data sets, and data stream model is more feasible to analyze network traffic and detect anomaly and attacks.

In this dissertation, we design effective techniques for detecting and attributing cyber criminals. We consider two kinds of fundamental techniques: forensics-sound attack monitoring and traceback, and forensics-sound online fraud detection. The contributions of our research are as follows: We propose several innovative algorithms which answer some open problems in fundamental statistics estimation over sliding windows. Those algorithms can be used to detect anomaly and attacks in networks. We also propose efficient and effective algorithms which can trace back stepping stone attacks and single packet attacks. Streaming algorithms are presented to detect click fraud in pay-per-click streams of online advertising networks.

CHAPTER 1. OVERVIEW

1.1 Introduction

With the phenomenal growth of the Internet, more and more people enjoy and depend on the convenience of its provided services. The Internet has spread rapidly to almost all over the world. Up to June 2008, the Internet has distributed to over 233 countries and world regions, and has more than 1.46 billion users [1]. Unfortunately, the wide use of computer and Internet also has opened doors to cyber attackers. There are different kinds of attacks that an end user of a computer or Internet has to face. For instance, there may be various viruses on the hard disk, there may be several backdoors opened in the operating system, and there may be a lot of phishing e-mails in his/her mailbox. Also, more and more fraud activities appear in online advertising networks and online auction systems. According to the 2008 CSI computer crime & security survey by Computer Security Institute (CSI) [91], cyber attacks cause a lot of money losses each year. Network attackers can easily hide their identities through IP spoofing, stepping stones, network address translators (NATs), Mobile IP or other ways, and thereby reduce the chance of being captured. The current IP network infrastructure lacks measures and cannot effectively deter and identify motivated and well-equipped attackers. Therefore, innovative traceback schemes are required to attribute the real attackers. By the way, network traffic always comes with high rate in distributed format without obvious beginning and ending. These properties make network traffic much different compared with traditional data sets, and data stream model is more feasible to analyze network traffic.

In our research, we study and design efficient and effective techniques for detecting and attributing cyber criminals. We generally consider two kinds of fundamental techniques:

forensics-sound[1] attack monitoring and traceback, and forensics-sound online fraud detection. We hope that our work may serve as fundamental components which can be widely applied in network security and many other domains.

1.2 A Motivating Scenario

As shown in Figure 1.1, the Internet consists of many autonomous systems (ASes). To defend potential attacks, many monitors are distributed in the network. They generate a lot of log information which will be sent to one or more network operations centers (NOCs).

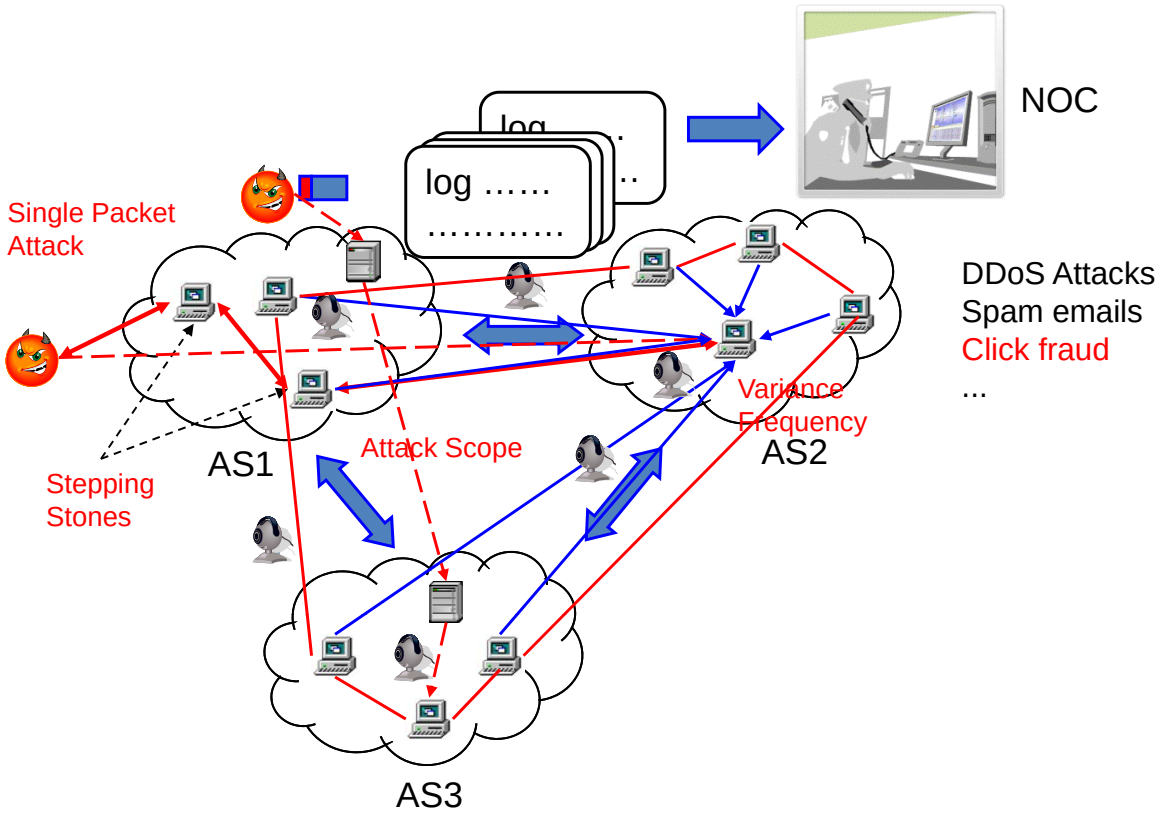


Figure 1.1 A Motivating Scenario

Suppose that a host is under a Distributed Denial-of-Service (DDoS) attack. If we take a look at the corresponding log information, its statistics, such as variance and frequency, may change a lot compared with its normal values. Also, suppose that a worm is spreading over the

[1] "Forensics-sound" means acceptable effectiveness (correctness and error bound) and efficiency (space and time).

Internet, and many machines are affected. The NOC may require a worm spreading map, so that it can use the geometry information to take some countermeasures, for instance, shutting down some affected machines, or requiring the corresponding monitors with in the affected scope to report more detailed information.

There are many kinds of attacks in the Internet. Consider that there is an attacker which launches attacks through several intermediate hosts instead of connecting to the victim directly. This kind of attacks is called stepping stone attack. Such an indirect attack is difficult to attribute. Stepping stone attacks are interactive attacks which generate a lot of IP packets. On the contrary, some attacks can be finished within a well-targeted packet, which are called single packet attacks.

Furthermore, the final purposes of the attackers to intrude and control the others' machines are not only for fun, but also to do many malicious tasks. They can control thousands or even millions of bots to launch DDoS attacks, to send out spam emails, to generate fraudulent clicks or eBay transactions to earn money, etc.

To defend such an attack scenario, several potential queries should be answered:

- Is there any anomaly in the network, in terms of frequency, variance, etc.?
- Can we find the geometry scope of the attackers or affected hosts?
- If the attackers are several hops away, how to attribute them?
- If the attacks are finished within one packet, how to trace back the attackers?
- How to detect fraudulent clicks?

If we like, many other queries or requirements can be listed here. We are interested in the queries within two categories:

(1) How to analyze network traffic to detect anomaly and attacks, and trace back to the real attackers?

(2) How to defend the fraud activities from attackers?

1.3 Objectives

In our research, we propose to design efficient and effective techniques for attributing and defending against cyber criminals. Our research goals are

- To detect network anomaly and attacks, and trace back attacks to real sources.
- To defend online fraud activities.

To achieve these goals, we studied two kinds of fundamental techniques as follows.

1.3.1 Forensics-Sound Attack Monitoring and Traceback Techniques

1.3.1.1 Attack Monitoring Techniques

To monitor and detect anomaly and attacks in networks, we need to process a lot of data gathered in the networks, such as logs and other information. If we treat network traffic as a data set, and use traditional database system techniques to analyze it, we really can detect many anomalies and attacks. However, network traffic has its own properties which make it much different compared with traditional data sets.

(1) Network traffic always increases much faster than the development of computing technology (CPU, storage devices, etc).

It means that to analyze the network traffic, we always meet the problem of high rate/volume data vs. limited processing resources. In each second, high rate network traffic is generated, and a lot of logged activities wait to be analyzed. In many cases it is impossible to record all traffic for later analysis with limited resources.

(2) Network traffic usually has no obvious beginning and ending.

It implies that it is improper to store data first and analyze later, or to cut data into pieces and analyze them separately.

(3) Current status of network traffic is usually more important than previous status.

It implies that when analyzing network traffic, the data with different timestampes should have different weights.

(4) Network traffic should be analyzed in online mode to guarantee quick response.

If we have the capabilities to analyze the network traffic in online mode, we can identify the ongoing attacks and thus take countermeasures to eliminate or mitigate the potential damages.

(5) The analysis results should be precise enough.

If there are a lot of errors in the network analysis results, then the network will be filled with false alarms, which are very annoying and may disturb the normal daily operations of the network.

Therefore, analyzing network traffic should have a different model compared with the processing of traditional data sets. Data stream model is introduced in 1998 [56], which has received considerable attention [18, 82]. In data stream model, the assumption is that the input data is continuous with high rate such that the processing system has no enough space to hold them in memory and analyze them later. That is, the interested characteristics must be gathered with only one pass. Therefore, an algorithm over data stream model is desired to have all or part of the following capabilities:

(1) Polynomial space requirement with respect to the size of the input data stream.

(2) One pass processing.

(3) More weight on recent data.

(4) Short processing time for each input element, and short computational time to answer queries.

(5) Error-bounded.

To put more weight on recent data, sliding window model is a general time decay model for data streams, which only considers the most recent N elements evenly, and discards the influence of all older elements, where N is the window size. Most available algorithms/techniques in database system are not perfectly qualified in data stream model. In other words, analyzing network traffic using the same techniques as analyzing a typical database will meet problems, and new techniques are required.

Variance estimation can be useful in many different research fields, e.g., network monitoring, intrusion detection, financial monitoring, weather forecast, disaster forecast, etc. Variance is often related to anomaly and status change which is a powerful measure for anomaly detec-

tion and forecast. **Frequency** is a fundamental characteristics in many database applications. Furthermore, it has many applications in network security. For instance, it can be used to solve the heavy-hitter and super-spreader problem. Here, a heavy-hitter means a source IP which sends out a lot of packets; A super-spreader is a source IP which links to a lot of destinations and may be a network scanner. Although some solutions have been proposed, there are still gaps between the bounds of available algorithms and theoretical lower bounds, in terms of space requirement or processing time. Consequently, our goal is to find optimal variance and frequency estimation algorithms over sliding windows.

Geometric computation has many applications, such as computer graphics, computer-aided design and manufacturing (CAD/CAM), geographic information systems (GIS), integrated circuit geometry design and verification, etc. Also, geometric computation becomes an important issue in network security after distributed networks (e.g., sensor networks) are widely researched and deployed. For instance, in the early age of worm propagation, after receiving thousands of alarms from distributed network monitors, a geometric map is necessary to show which regions have been affected so that countermeasures can be executed to interrupt the worm propagation. **Diameter** can scale how far the worm has propagated, and **convex hull** can reflect the boundary of the affected hosts. Furthermore, geometric computation is required not only in the domain of geometric coordinates, but also in many other fields in network security. For instance, network logs contain a lot of hosts which volume, connection numbers, etc. are recorded, and we want to analyze and detect the dominant hosts in some terms which should have more chance to be attackers. The **skyline** calculation can be applied to this purpose. Our goal is to provide efficient geometric computation algorithms over data stream models.

1.3.1.2 Attack Traceback Techniques

Network-based attackers can easily hide their identities through IP spoofing, stepping stones, network address translators (NATs), Mobile IP or other ways, and thereby reduce the chance of being captured. However, the current IP network infrastructure cannot effec-

tively deter and identify motivated and well-equipped attackers. Therefore, it is desirable to design effective and efficient traceback systems to attribute attackers and help reconstruct cyber crime scenes. There are many forms of network-based attacks. In our research, we consider two types of network attacks: stepping stone attacks and single packet attacks.

(1) Stepping Stone Attack Traceback

Network based attackers often relay attacks through intermediary hosts (i.e., stepping stones) to evade detection. In addition, attackers make detection more difficult by encrypting attack traffic and introducing delay and chaff perturbations into stepping stone connections. Several approaches have been proposed to detect stepping stone attacks. However, none of them performs effectively when delay and chaff perturbations exist simultaneously. In our work, we propose correlation algorithms to efficiently and effectively attribute stepping stone attacks when delay and chaff perturbations exist simultaneously.

(2) Single Packet Attack IP Traceback

While most DDoS attacks are conducted by flooding networks with large amounts of traffic, there are attacks which require significantly smaller packet flows. Some attacks (e.g., Teardrop) can even succeed by using only one well-targeted packet. Although several IP traceback systems have been designed to trace back single packet attacks, their effectiveness is restrained by high false positive rates. In this work, we design a Bloom filter-based topology-aware single packet IP traceback system which has reasonable space requirement and fast processing speed, and simultaneously has forensics-sound tracing capabilities.

1.3.2 Forensics-Sound Online Fraud Detection Techniques

With the rapid growth of the Internet, online advertisement plays a more and more important role in the advertising market. One of the current and widely used revenue models for online advertising involves charging for each click based on the popularity of keywords and the number of competing advertisers. This pay-per-click model leaves room for individuals or rival companies to generate false clicks (i.e., click fraud), which pose serious problems to the development of healthy online advertising market.

To detect click fraud, an important issue is to detect duplicate clicks over decaying window models, such as jumping windows and sliding windows. Decaying window models can be very helpful in defining and determining click fraud. However, although there are available algorithms to detect duplicates, there is still a lack of practical and effective solutions to detect click fraud in pay-per-click streams over decaying window models. We propose to address the problem of detecting duplicate clicks in pay-per-click streams.

Generally speaking, we hope that our proposed work can serve as fundamental components which can be widely utilized in network security and many other domains. These forensics-sound attack monitoring techniques, traceback techniques, and fraud detection techniques can be integrated together to develop a holistic approach in detecting and attributing attacks/criminals such as DDoS, botnet, click-fraud, on-line auction fraudsters, etc.

1.4 Contributions

The contributions of our research are as follows:

(1) We propose several innovative algorithms which answer some open problems in fundamental statistics estimation over sliding windows. Those algorithms can be used to detect anomaly and attacks in networks. We also propose efficient and effective algorithms which can trace back stepping stone attacks and single packet attacks.

- We address the problem of maintaining ϵ-approximate variance of data streams over sliding windows. To our knowledge, the best existing algorithm requires $O(\frac{1}{\epsilon^2}\log N)$ space, though the lower bound for this problem is $\Omega(\frac{1}{\epsilon}\log N)$. We propose the first ϵ-approximation algorithm to this problem that is optimal in both space and worst case time. Our algorithm requires $O(\frac{1}{\epsilon}\log N)$ space. Furthermore, its running time is $O(1)$ in worst case.

- We study the problem of estimating ϵ-approximate frequency in data streams over sliding windows. We propose the first efficient deterministic algorithm which can achieve $O(\frac{1}{\epsilon})$ space requirement and only need $O(1)$ running time to process each item in the

data stream and to answer a query. We present two novel deterministic algorithms, SNAPSHOT-BASIC and SNAPSHOT-ADVANCED, which need $O(\frac{1}{\epsilon})$ space. Furthermore, SNAPSHOT-ADVANCED only needs $O(1)$ running time. Both theoretical and experimental analysis show that our algorithms can be adopted to estimate item frequencies in sliding windows over on-line high-rate data streams with the error guarantee. In addition, as an application of our algorithms, we extend them to solve the problem of estimating flow size.

- we study the problem of estimating ϵ-approximate diameter, convex hull and skyline in data streams over sliding windows. To our knowledge, the best existing algorithm for the problem of estimating ϵ-approximate diameter in data streams over sliding windows requires $O((\frac{1}{\epsilon})^{\frac{d+1}{2}} \log \frac{R}{\epsilon})$ space [28], where R is the ratio between the largest distance and the smallest distance of a pair of points, and d is the dimension. We first present an improved algorithm which only requires $O((\frac{1}{\epsilon})^{\frac{d+1}{2}} \log R)$ space. We then extend our algorithm to solve convex hull estimation problem over sliding windows, and prove that the exact diameter algorithm can get the ϵ-approximate convex hull estimation directly. Finally, we propose a novel algorithm to estimate skyline which requires $O(\frac{1}{\epsilon^d} \log \epsilon R)$ space.

- We propose and analyze algorithms which represent that stepping stone attackers cannot always evade detection only by adding limited delay and independent chaff perturbations. We provide the upper bounds on the number of packets needed to confidently detect stepping stone connections from non-stepping stone connections with any given probability of false attribution. We compare our algorithms with previous ones and the experimental results show that our algorithms are more effective in detecting stepping stone attacks in some scenarios.

- We propose a topology-aware single packet IP traceback system, namely *TOPO*. We utilize router's local topology information, i.e., its immediate predecessor information. Our performance analysis shows that TOPO can reduce the number and scope of unnecessary

queries, and significantly decrease false attributions. Furthermore, to improve the practicability of Bloom filter-based IP traceback systems, we design TOPO to allow partial deployment while maintaining its traceback capability. When Bloom filters are used, it is difficult to decide their optimal control parameters a priori. We design a k-adaptive mechanism which can dynamically adjust parameters of Bloom filters to reduce the false positive rate.

(2) We propose streaming algorithms to detect click fraud in pay-per-click streams of online advertising networks.

- We address the problem of detecting duplicate clicks in pay-per-click streams over jumping windows and sliding windows, and are the first that propose two innovative algorithms that make only one pass over click streams and require *significantly less* memory space and operations. GBF algorithm is built on *group Bloom filters* which can process click streams over jumping windows with small number of sub-windows, while TBF algorithm is based on a new data structure called *timing Bloom filter* that detects click fraud over sliding windows and jumping windows with large number of sub-windows. Both GBF algorithm and TBF algorithm have zero false negative. Furthermore, both theoretical analysis and experimental results show that our algorithms can achieve low false positive rate when detecting duplicate clicks in pay-per-click streams over jumping windows and sliding windows. A patent [55] is pending based on our research.

1.5 Dissertation Organization

The rest of this dissertation is structured as follows: Chapter 2 provides a review of literature for the problems targeted in our research. We present our research in data streaming techniques in Chapter 3, and these techniques can be used to monitor attacks. Attack traceback algorithms are proposed in Chapter 4. Chapter 5 discusses our techniques in online fraud detection. We finally summarize our research and discuss our future work in Chapter 6.

CHAPTER 2. LITERATURE REVIEW

This chapter reviews recent literatures which correlates to our research in this dissertation.

2.1 Data Processing over Data Stream Models

Data stream is a hot topic in many research areas, including databases, geometric computation, network security, etc. After this concept is first introduced by Henzinger et al. in 1998 [56], many classical problems are reconsidered under this model. More information can be found in surveys [18] and [82].

2.1.1 Variance Estimation

Datar et al. [35] proposed an algorithm to solve the *Basic-Counting* problem over sliding windows using *Exponential Histograms* (EH). Given the sliding window size N, and the relative error ϵ, the space taken by their algorithm is $O(\frac{1}{\epsilon}\log N)$, and the time taken to process each element is $O(1)$ amortized and $O(\log N)$ in worst case. Gibbons and Tirthapura [50] presented an algorithm based on a data structure called the *wave* that uses the same space as in [35]. However, their algorithm only takes $O(1)$ time to process each element in worst case. They also presented three extensions of the sliding window model for the distributed streams.

The EH algorithm [35] can estimate a class of aggregate functions over sliding windows. It applies to any function f satisfying the following properties for all multisets X, Y:

1. $f(X) \geq 0$.

2. $f(X) \leq poly(|X|)$.

3. $f(X \bigcup Y) \geq f(X) + f(Y)$.

4. $f(X \bigcup Y) \leq C_f(f(X) + f(Y))$, where constant $C_f \geq 1$.

However, variance does not satisfy the fourth property. For instance, suppose both X and Y have zero (or very small) variance, but different means. Then the variance of $X \bigcup Y$ may not be bounded by any constant C_f. Therefore, the EH algorithm cannot be utilized in variance estimation directly. Consequently, Babcock et al. [17] proposed a new algorithm for maintaining variance over sliding windows of a data stream. In their algorithm, elements in the data stream are partitioned into buckets, and the synopses of these buckets are maintained, which can be used to merge adjacent buckets to save space according to particular combination constraints. This algorithm requires $O(\frac{1}{\epsilon^2}\log N)$ space, and its running time is $O(1)$ amortized and $O(\frac{1}{\epsilon^2}\log N)$ in worst case. However, as mentioned by the authors of [17], there is a gap of $\frac{1}{\epsilon}$ between the optimal bound $\Omega(\frac{1}{\epsilon}\log N)$ and their upper bound.

Besides the bit counting and variance problems, several other problems of capturing characteristics of data streams over sliding windows are studied. Zhu and Shasha [121] introduced *basic windows* to compute the statistics over a sliding window. Golab et al. [51] utilized the basic windows mechanism in their deterministic algorithm to identify frequent items over sliding windows. Babcock et al. [17] proposed an algorithm for maintaining a k-median clustering over sliding windows of a data stream. Feigenbaum et al. [45] investigated the diameter problem in the streaming and sliding-window models, and the diameter is the maximum distance between a pair of elements within a data stream. Lin et al. [67] considered the problem of continuously maintaining ϵ-approximate quantiles over sliding windows in a data stream. Arasu and Manku [16] studied the problem of maintaining ϵ-approximate counts and quantiles over sliding windows in a data stream. Tirthapura et al. [104] considered the problem of maintaining sketches of recent elements of a asynchronous data stream in which the observed order of data is not the same as the time order in which the data was generated.

2.1.2 Frequency Estimation

2.1.2.1 Frequency Statistics Algorithms

Problems related to frequency estimate have been studied by many researchers in different fields. Flajolet and Martin [46] and Whang et al. [109] proposed probabilistic algorithms to

estimate the number of distinct items in a large collection of data in a single pass. Alon et al [14] researched the frequency moments of a sequence of items. Gibbons and Matias [49] proposed sampling algorithms to identify top-k queries. Fang et al. [44] presented several algorithms based on hashing to compute iceberg queries, but each requires at least two passes over the data stream. Estan and Varghese [41] presented a sampling algorithm named *sample and hold* and a hash-based algorithm named *multistage filters*. Charikar et al. [30] proposed a data structure called *count sketch* to find frequent items. Manku and Motwani [72] presented a randomized algorithm for computing frequency counts exceeding a user-specified threshold. Cormode and Muthukrishnan [34] proposed randomized algorithms that identify frequent items when both insertions and deletions are present. Manjhi et al. [70] studied the problem of computing frequency counts for items occurring frequently in the union of multiple distributed data streams. Super-spreader problem is researched by Estan et al. [42], Venkataraman et al. [105] and Zhao et al. [120]. Other recent works on data stream algorithms have been surveyed in [18, 82].

Here we shortly introduce the algorithm proposed by Misra and Gries [80] which maintains ϵ-approximate frequency estimation using exact $\frac{1}{\epsilon}$ counters with one pass. Suppose there is a data stream of N items. First, a list of $\frac{1}{\epsilon}$ counters is initiated to 0. For each item in the stream, we increase its corresponding counter (if any) by 1. If it has no corresponding counter and there is a counter cnt_x which equals zero, then cnt_x is assigned to this item and set to 1. If the item has no corresponding counter and there is no zero counter, then every counter is decreased by 1. Misra-Gries algorithm can maintain ϵ-approximate frequency estimation, because the decrease operation of the $\frac{1}{\epsilon}$ counters can only happen at most $\frac{N}{(1/\epsilon)+1} < \epsilon N$ times. Karp et al. [61] and Demaine et al. [37] improved the processing time of Misra-Gries algorithm to $O(1)$ in the worst case. Metwally et al. [76] proposed an algorithm which can report frequent items and top-k items simultaneously.

2.1.2.2 Algorithms over Sliding Windows

The previous frequency statistics algorithms [14, 30, 34, 41, 42, 44, 46, 49, 70, 72, 105, 109, 120] are designed to process all or a piece of the data stream. They cannot be directly used to process data streams over sliding windows.

Zhu and Shasha [121] introduced *basic windows* to compute the statistics over sliding windows. A sliding window is subdivided equally into shorter basic windows. Only a synopsis data structure is maintained for each basic window and the entire sliding window. When a new basic window is created, its synopsis is added to that of the entire sliding window, and the synopsis of the oldest window is expired and deleted. The "smooth" of the sliding window depends on the size of the basic windows. Smaller sized basic windows have finer scale but need more space. Larger sized basic windows have coarse scale but need less space. Golab et al. [51] utilized the basic windows mechanism in their deterministic algorithm to identify frequent items over sliding windows. Their algorithm identifies and stores the exact top-k frequent items in each basic window. Then these local top-k frequent items are used to find and update the global frequent items. To find the exact top-k frequent items in the current basic window, this algorithm keeps a counter for each unique item. In the worst case, all items in the current basic window may be unique, therefore this algorithm needs $sizeof$(basic windows) entries in the worst case.

Datar et al. [35] proposed an algorithm to solve the *Basic-Counting* problem over sliding windows using *Exponential Histograms* (EH). Given the sliding window size of N, and the relative error ϵ, the space taken by their algorithm is $O(\frac{1}{\epsilon}\log N)$, and the time taken to process each item is $O(1)$ amortized and $O(\log N)$ in worst case. Gibbons and Tirthapura [50] presented an algorithm based on a data structure called the *wave* that uses the same space as in [35]. However, it only takes $O(1)$ time to process each item in the worst case. Lee and Ting [64] considered the *significant one counting* problem over sliding windows, where only the items which numbers exceed a user-specified threshold are maintained.

Babcock et al. [17] proposed algorithms for maintaining variance and maintaining a k-median clustering over sliding windows of a data stream. Zhang and Guan [115] also considered

how to efficiently maintain variance in sliding windows, and proposed an algorithm which is optimal in terms of both space requirement and worst case running time. Feigenbaum et al. [45] and Chan and Sadjad [28] investigated the diameter problem in the streaming and sliding-window models. The diameter is the maximum distance between a pair of items within a data stream. Lin et al. [67] considered the problem of continuously maintaining ϵ-approximate quantiles over sliding windows in a data stream.

Arasu and Manku [16] studied the problem of maintaining ϵ-approximate counts and quantiles over a sliding window in a data stream. Their deterministic algorithm for approximate counts uses the Misra-Gries algorithm [80] as a black-box. They constructed many black-boxes with different sizes in different levels. These black-boxes can answer the queries combined together. This algorithm requires $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$ space and $O(\log\frac{1}{\epsilon})$ running time. Lee and Ting [65] studied the same problem and proposed an algorithm which achieves $O(\frac{1}{\epsilon})$ space. However, their algorithm needs $O(\frac{1}{\epsilon})$ processing time for update and query.

Tirthapura et al. [104] considered the problem of maintaining sketches of recent elements of an asynchronous data stream in which the observed order of data is not the same as the time order in which the data was generated. They presented algorithms for maintaining sketches of all elements within the sliding timestamp window that can give provably accurate estimates of two basic aggregates, the sum and the median, of a stream of numbers. Busch and Tirthapura [26] proposed a deterministic algorithm for this scenario recently.

2.1.3 Geometric Estimation

Problems related to geometric computation have been studied by many researchers in different fields. Also, a lot of sliding window related algorithms were proposed recently.

2.1.3.1 Diameter Algorithms

For diameter problem, Preparata and Shamos provideed an $O(n\log n)$ algorithm in two-dimension [88], where n is the number of points. Clarkson and Shor proposed a randomized algorithm in three-dimension using $O(n\log n)$ time [32], and later an optimal deterministic

algorithm was designed by Ramos [89].

However, for higher dimensions, there is lack of subquadratic algorithms for diameter computation. Therefore, several approximation algorithms were proposed. Agarwal et al. [13] presented an approximation algorithm using $O((\frac{1}{\epsilon})^{\frac{d-1}{2}})$ space and $O((\frac{1}{\epsilon})^{\frac{d-1}{2}}n)$ time. Hershberger and Suri [57] provided an adaptive sampling algorithm in two-dimension which uses $O((\frac{1}{\epsilon})^{\frac{1}{2}})$ space and $O(\log \frac{1}{\epsilon})$ time per point. Indyk [59] developed a c-approximation algorithm using $O(dn^{\frac{1}{c^2-1}})$ space and running time for $c > \sqrt{2}$. Chan [27] proposed a fast ϵ-approximation algorithm which runs in $O(n + (\frac{1}{\epsilon})^{d-1.5})$ time.

2.1.3.2 Convex Hull Algorithms

Richardson [92] proposed an algorithm to approximate two-dimensional static convex hull scaled by Hausdorff distance using $O((\frac{1}{\epsilon})^{\frac{1}{2}})$ space. Cormode and Muthukrishnan [33] presented a radial histogram algorithm which can estimate a two-dimensional convex hull using $O(\frac{1}{\epsilon})$ space and $O(1)$ running time per point. Agarwal et al. [12] presented a data structure to maintain ϵ-approximate convex hull using $O((\frac{1}{\epsilon})^{\frac{1}{2}} \log n)$ space and $O((\frac{1}{\epsilon})^{\frac{1}{2}})$ running time per point. Hershberger and Suri's adaptive sampling algorithm [57] also maintains two-dimensional convex hull estimation using $O((\frac{1}{\epsilon})^{\frac{1}{2}})$ space and $O(\log \frac{1}{\epsilon})$ time per point. However, their algorithms may not work over sliding windows.

2.1.3.3 Skyline Algorithms

Skyline computation was first studied by Kung et al. [63] in computational geometry. After that, skyline computation and its variants have been well studied. Tan et al. [102] proposed the first progressive technique that can output skyline points without scanning the whole dataset in advance. In the database context, Borzsonyi et al. [19] developed two solutions based on divide-and-conquer (DC) and blocknested-loop (BNL), respectively. Kossmann et al. [62] and Papadias et al. [84] presented progressive algorithms based on the nearest neighbor search.

2.1.3.4 Sliding Windows Algorithms

Most previous geometric computation algorithms [13, 19, 27, 32, 33, 57, 59, 62, 63, 84, 88, 89, 92, 102] are designed to process all or a piece of the data stream. Therefore, they cannot be directly used to process a stream of points over sliding windows.

Datar et al. [35] proposed an algorithm to solve the *Basic-Counting* problem over sliding windows using *Exponential Histograms* (EH). After this paper, a lot of algorithms are proposed to solve different statistics problems over sliding windows, such as variance, quantile, k-median clustering, frequency, etc. [16, 17, 26, 50, 65, 67, 104, 115, 116, 117]. More inforamtion can be found in surveys [18] and [82].

For the problem of diameter estimation in two-dimension over sliding windows, Feigenbaum et al. [45] proposed a clustering algorithm which uses $O(\frac{1}{\epsilon}\log^2 N \log(\frac{R}{\epsilon}))$ space. Chan and Sadjad [28] proposed an optimal algorithm for one-dimension which uses $\Theta(\frac{1}{\epsilon}\log R)$ space. For higher dimensions, their algorithm needs $O((\frac{1}{\epsilon})^{\frac{d+1}{2}}\log\frac{R}{\epsilon})$ space.

Lin et al. [68] and Tao and Papadias [103] considered the skyline computation problem in a data stream over sliding windows. However, they studied this problem in a different context. In [68], the estimation error is not theoretically guaranteed. The algorithm in [103] computes the exact skyline which requires huge space in worst case when the window size is large.

2.2 Attack Traceback

2.2.1 Stepping Stone Attack Attribution

Ever since the problem of detecting stepping stones was first proposed by Staniford-Chen and Heberlein [98], several approaches have been proposed to detect encrypted stepping stone attacks [54].

The ON/OFF based approach proposed by Zhang and Paxson [119] is the first timing-based method which can trace stepping stones even if the traffic were to be encrypted. In their approach, they calculated the correlation of different flows by using each flows' OFF periods. A flow is considered to be in an OFF period when there is no data traffic on a flow for more than

a time period threshold. Their approach comes from the observation that two flows are in the same connection chain if their OFF periods coincide. We refer to this method as ***ON/OFF***.

Yoda and Etoh [113] presented a deviation based approach for detecting stepping stone connections. The deviation is defined as the difference between the average propagation delay and the minimum propagation delay of two connections. This scheme comes from the observation that the deviation for two unrelated connections is large enough to be distinguished from the deviation of connections in the same connection chain. We refer to this method as ***Deviation***.

Wang et al. [107] proposed a correlation scheme using inter-packet delay (IPD) characteristics to detect stepping stones. They defined their correlation metric over the IPDs in a sliding window of packets of the connections to be correlated. They showed that the IPD characteristics may be preserved across many stepping stones. We refer to this method as ***IPD***.

Wang and Reeves [106] presented an active watermark scheme which is designed to be robust against certain delay perturbations. The watermark is introduced into a connection by slightly adjusting the inter-packet delays of selected packets in the flow. If the delay perturbation is not quite large, the watermark information will remain along the connection chain. This is the only active stepping stone attribution approach. We refer to this method as ***Watermark***.

Strayer et al. [100] presented a State-Space algorithm which is derived from their work on wireless topology discovery. When a new packet is received, each node is given a weight which decreases as the elapsed time from the last packet from that node increases. Then the connections on the same connection chain will have higher weights than other connections. We refer to this method as ***State-Space***. This approach is based on two assumptions. First, the likelihood of one transmission being a response to a prior transmission generally decreases as the elapsed time between these transmission increases. Second, the inter-arrival times between a fixed event and any other event are approximately Poisson distributed.

However, none of these previous approaches can effectively detect stepping stones when

delay and chaff perturbations exist simultaneously. When chaff perturbation is added to the flows and no delay perturbation exists, the timing information used in *deviation*, *IPD* and *watermark* may be destroyed entirely. Also, many OFF periods will disappear, which degrades the performance of the *ON/OFF* method. Even though the *State-Space* still achieves 0% false negative rate, if the delay perturbation is an unknown constant, the false negative rate of *State-Space* approach will obviously increase. Therefore, if both delay and chaff perturbations exist simultaneously, none of the previous approaches may provide good performance.

Although no experimental data is available, Donoho et al. [39] indicated that there are theoretical limits on the ability of attackers to disguise their traffic using evasions for sufficiently long connections. They assumed that the intruder has a maximum delay tolerance, and used wavelets and similar multiscale methods to separate the short-term behavior of the flows (delay or chaff) from the long-term behavior of the flows (the remaining correlation). However, this method requires the intrusion connections to remain for long periods, and the author never experimented to show the effectiveness against chaff perturbation. We refer to this method as ***Multiscale***. These evasions consist of local jittering of packet arrival times and the addition of superfluous packets.

Blum et al. [23] proposed and analyzed algorithms for stepping stone detection using ideas from Computational Learning Theory and the analysis of random walks. They achieved provable (polynomial) upper bounds on the number of packets needed to confidently detect and identify stepping stone flows with proven guarantees on the false positives, and provided lower bounds on the amount of chaff that an attacker would have to send to evade detection. However, their upper bounds on the number of packets required is large, while the lower bounds on the amount of chaff needed for attacker to evade detection is very small. They did not discuss how to detect stepping stones without enough packets or with large amounts of chaff, and did not show experimental results. We refer to their methods as ***Detect-Attacks*** and ***Detect-Attacks-Chaff***.

2.2.2 IP Traceback

In this section, we review major existing IP traceback schemes that have been designed to attribute the origin of IP packets through the Internet. We roughly categorize them into four primary classes: a) Active Probing [25, 99], b) ICMP Traceback [21, 71, 110], c) Packet Marking (including deterministic, probabilistic, and algebraic packet marking) [20, 36, 85, 93, 96], and d) Log-based Traceback [66, 74, 94, 95].

2.2.2.1 Active Probing

Stone [99] proposed a traceback scheme called *CenterTrack*, which selectively reroutes packets in question directly from edge routers to some special tracking routers. The tracking routers determine the ingress edge router by observing from which tunnel the packet arrives. This approach requires the cooperation of network administrators and the management overhead is considerably large.

Burch and Cheswick [25] outlined a technique for tracing spoofed packets back to their actual source without relying on the cooperation of intervening ISPs. The victim actively changes the traffic in particular links and observes the influence on attack packets, and thus can determine where the attack comes from. This technique cannot work well on distributed attacks, and requires the attacks remain active during the time period of traceback.

2.2.2.2 ICMP Traceback (iTrace)

Bellovin [21] proposed a scheme named *iTrace* to traceback using ICMP messages for authenticated IP marking. In this scheme, each router samples (with low probability) the forwarding packets, copies the contents into a special ICMP traceback message, adds its own IP address as well as the IP of the previous and next hop routers, and forwards the packet either to the source or destination address. By combining the information obtained from several of these ICMP messages from different routers, the victim can then reconstruct the path back to the origin of the attacker.

A drawback of this scheme was that it is much more likely that the victim will get ICMP

messages from routers nearby than from routers farther away. This implies that most of the network resources spent on generating and utilizing iTrace messages will be wasted. An enhancement of iTrace, called *"Intention-Driven iTrace"*, was proposed in [71, 110]. By introducing an extra "intention-bit", it is possible for the victim to increase the probability of receiving iTrace messages from remote routers.

2.2.2.3 Packet Marking

Savage et al. [93] proposed a *Probabilistic Packet Marking* (PPM) scheme. Thereafter, several other PPM-based schemes have been developed [96, 85, 36]. The baseline idea of PPM is that routers probabilistically write partial path information into the packets during forwarding. If the attacks are made up of a sufficiently large number of packets, eventually, the victim may get enough information by combining a modest number of marked packets to reconstruct the entire attack path. This allows victims to locate the approximate source of attack traffic without requiring outside assistance.

Deterministic Packet Marking (DPM) scheme proposed by Belenky and Ansari [20] involves the marking of each individual packet when it enters the network. The packet is marked by the interface closest to the source of the packet on the edge ingress router. The mark remains unchanged as long as the packet traverses the network. However, there is no way to get the whole paths of the attacks.

Dean et al. proposed an *Algebraic Packet Marking* (APM) which reframes the traceback problem as a polynomial reconstruction problem and uses techniques from algebraic coding theory to provide robust methods of transmission and reconstruction. The advantage of this scheme is that it offers more flexibility in design and more powerful techniques that can be used to filter out attacker generated noise and separate multiple paths. But it shared similarity with PPM in that it requires a sufficiently large number of attack packets.

2.2.2.4 Log-based Traceback

The basic idea of log-based traceback is that each router stores the information (digests, signature, or even the packet itself) of network traffic through it. Once an attack is detected, the victim queries the upstream routers by checking whether they have logged the attack packet in question or not. If the attack packet's information is found in a given router's memory, then that router is deemed to be part of the attack path. Obviously, the major challenge in log-based traceback schemes is the storage space requirement at the intermediate routers.

Matsuda et al. [74] proposed a hop-by-hop log-based IP traceback method. Its main features are logging *packet feature* that is composed of a portion of the packet for identification purpose, and an algorithm using data-link identifier to identify the routing of a packet. However, for each received packet, about 60 bytes data should be recorded. The resulted large memory space requirement prevents this method from being applied to high speed networks with heavy traffic.

Although today's high-speed IP networks suggest that classical log-based traceback schemes would be too prohibitive because of the huge memory requirement, log-based traceback becomes attractive after Bloom filter-based (i.e., hash-based) traceback schemes were proposed. *Bloom filters* were presented by Burton H. Bloom [22] in 1970, and have been widely used in many areas such as database and networking [24]. A Bloom filter is a space-efficient data structure for representing a set of n elements to respond membership queries. It is a vector of m bits which are all initialized to value 0. Then each element is inserted into the Bloom filter by hashing it using k independent uniform hash functions with range $\{1, 2, ..., m\}$ and setting the corresponding k bits (some bits may be overlapped) in the vector to value 1. Given a query whether an element is present in the Bloom filter, we hash this element using the same k hash functions and check if all the corresponding bits are set to 1. If any one of them is 0, then undoubtedly this element is not stored in the filter. Otherwise, we would say that it is present in the filter, although there is a certain probability that the element is determined to be in the filter while it is actually not. Such false cases are called *false positives*. The space-efficiency of Bloom filters is achieved at the cost of a small acceptable false positive rate f. From [43], we

have

$$f = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{-kn/m})^k. \tag{2.1}$$

When m and n are given, f is minimized for

$$k = \ln 2 \times m/n, \tag{2.2}$$

Thus, we have

$$f = 2^{-k} \approx 0.6185^{\frac{m}{n}}. \tag{2.3}$$

Bloom filters were first introduced into IP traceback area by Snoeren et al. [95]. They built a system named *Source Path Isolation Engine* (SPIE) which can trace the origin of a single IP packet delivered by the network in the recent past. They demonstrated that the system is effective, space-efficient and implementable in current or next-generation routing hardware. Bloom filters are used in each SPIE-equipped router to record the digests of all packets it received in the recent past. The digest of a packet is exactly several hash values of its non-mutable IP header fields and the prefix of the payload. Strayer et al. [101] extended this traceback architecture to IP-v6.

Shanmugasundaram, et al. [94] proposed a *payload attribution system* (PAS) based on a *Hierarchical Bloom filter* (HBF). HBF is such a Bloom filter in which an element is inserted several times using different parts of the same element. Compared with SPIE which is a packet digesting scheme, PAS only uses the payload excerpt of a packet. It is useful when the packet header is unavailable.

Li et al. [66] proposed a Bloom filter-based IP traceback scheme that requires an order of magnitude smaller processing and storage cost than SPIE, thereby being able to scale to much higher link speed. The baseline idea of their approach is to sample and log a small percentage of packets, and 1 bit packet marking is used in their sampling scheme. Therefore, their traceback scheme combines packet marking and packet logging together. Their simulation results showed that the traceback scheme can achieve high accuracy, and scale well to a large

number of attackers. However, as the authors also pointed out, because of the low sampling rate, their scheme is no longer capable to trace one attacker with only one packet.

2.3 Online Fraud Detection

2.3.1 Duplicate Detection

Bloom filters were presented by Burton H. Bloom [22] in 1970, and have been widely utilized in many areas such as networking and database [24]. Bloom filters have been utilized to detect duplicates in databases. Besides the Bloom filter-based duplicate detecting algorithms, many other algorithms have been proposed in different research areas such as database systems, operating systems, computer architecture and network security, etc. The problem of exact duplicate detection is well studied, and many algorithms have been proposed. Please refer to [48] for more relevant references.

2.3.2 Online Advertising Fraud Detection

Reiter et al. studied the *hit shaving* problem [90]. They considered the referral revenue model that the referrers get payment for every click-through to target sites. A hit shaving is a practice that the target site undetectably omits to pay a referrer for some number of clicks. They proposed several approaches to enable referrers to monitor the number of clicks for which they should be paid.

Anupam et al. proposed a *hit inflation* attack on pay-per-click online advertising schemes [15]. Hit inflation is a kind of click fraud which is difficult to detect, where a referrer artificially inflate the customers' clicks to make illegal profit. Metwally et al. proposed an algorithm called *Streaming-Rules* to detect hit inflations in data streams [77]. They also built an algorithm called *Similarity-Seeker* to discover coalitions among advertising publishers [78], and a framework is outlined on a generic architecture [79].

Metwally et al. considered the problem of detecting duplicates in click streams [75]. They proposed a scheme over landmark windows which is a direct deployment of Bloom filters [22]. They also discussed how to detect duplicates over jump windows and sliding windows. To run

over sliding windows, they use a modification of Bloom filters named *Counting Bloom Filter*, which is similar to [43]. However, their solution must keep all active click identifications in memory to slide them out later after they expire.

Deng and Rafiei considered the problem of approximately eliminating duplicates in streams with a limited space [38]. They proposed a data structure named *Stable Bloom Filter*, which randomly evicts the stale information to release room for more recent elements. However, their randomly evicting mechanism introduces false negatives besides the inherent false positives of Bloom filters.

Gandhi et al. described a type of click fraud threat to Internet advertising called *badvertisement* [47]. This attack utilizes malicious JavaScript to publish sponsored advertisements on clients' web browsers invisibly. The authors proposed active and passive approaches to detect and prevent such kind of click fraud.

CHAPTER 3. RESEARCH IN ATTACK ATTRIBUTION PART I: MONITORING TECHNIQUES

3.1 Variance Estimation over Sliding Windows

3.1.1 Introduction

The problem of capturing characteristics of large data streams has received considerable attention [18, 82]. Many characteristics of large data streams, such as sum, mean, variance, diameter, frequency, quantile, top-k list (hot list), distribution, etc., have been widely studied. If we have sufficient large space and do not have time constraints, we can obtain any characteristics that we want precisely. However, the issue in processing large data streams is that in many cases we probably do not have enough space to keep each element in the data streams. That is, we have to gather the interested characteristics with one pass and we have no chance to go through the data streams again.

An even greater challenge is to process data streams over sliding windows. An algorithm can work over sliding windows if it can not only gather the data streams characteristics, but also update the characteristics when new data is inserted and old data is expired. Unfortunately, many previous data stream algorithms cannot work over sliding windows. In this research, we are interested in estimating element variance in data streams over sliding windows, while using as little space and operation as possible and making only one pass over the data streams.

3.1.1.1 Motivation

Variance estimation can be useful in many different research fields, e.g., network monitoring, intrusion detection, financial monitoring, weather forecast, disaster forecast, etc. Variance is an

important statistical measure to evaluate the variability of random variables. Also, variance is often related to anomaly and status change, therefore variance is a powerful measure for anomaly detection and forecast. For instance, in weather forecast, variance is an important variable to accurately forecast precipitation, temperature, wind direction and speed, etc. In disaster forecast and reduction field, before disasters, such as earthquake, tornado and flood, happen, there usually exists some anomaly that can be observed and evaluated using variance.

To calculate the variance of a fixed series of elements is trivial. However, in many cases, different data have different weights (importance) according to the time. That is, there is time decay in the data streams. For instance, for a network intrusion detection system, the current network status is more important than that of one day ago, because we can use the current network status to detect ongoing anomaly and intrusions and accordingly take some measures to reduce the potential damage. Sliding window model is a general time decay model that only considers the most recent N elements evenly, and discards the influence of all older elements, where N is the window size. Sliding window model can provide recent data information by removing the stale data. Unfortunately, it is nontrivial to calculate the variance of data streams in sliding window model. For example, when a computer network is under attacks, it may show large variance in terms of connection numbers, traffic volume and delays, etc. However, it is difficult to maintain accurate variance of these variables over sliding windows when the network has heavy traffic.

Recently several algorithms are proposed for capturing different kinds of characteristics of large data streams over sliding windows [16, 17, 35, 45, 50, 51, 67, 104, 121]. For the problem of variance estimation over sliding windows, the best algorithm was proposed by Babcock et al. [17]. However, as mentioned by the authors of [17], there is still a gap between the optimal space bound and their upper bound. Also, the running time of their algorithm is not constant in worst case. Consequently, it is desired to find an efficient variance estimation algorithm over sliding windows which needs less space and runs in constant time.

3.1.1.2 Problem Definition

Definition 1. ***Variance:*** *Let $\{x_1, x_2, \cdots, x_N\}$ be a series of N integers (which may be negative). The variance of these N numbers is defined by*

$$V = \sum_{i=1}^{N}(x_i - \mu)^2,$$

*where $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$ denotes the **mean** of these N integers.*

To calculate the exact variance of fixed N numbers is trivial. We only need to keep two variables: one is the sum of all numbers, and the other is the sum of all numbers' squares. However, calculating exact variance over sliding windows is more difficult and challenging. For a stream of numbers, if we want to calculate the exact variance of the most recent N numbers, the memory requirement is $\Omega(N)$. In some cases, it is impossible to keep so much memory space for variance estimation purpose. Consequently, we have to find efficient and accurate approximation algorithms to estimate the variance over sliding windows.

Definition 2. ***ϵ-Approximation:*** *Let $\hat{V}$ denote the estimation of the real value V. $\hat{V}$ is an ϵ-approximation of V if*

$$|V - \hat{V}| \leq \epsilon V,$$

where $0 < \epsilon < 1$ denotes the relative error.

Problem Statement In this research, we study the problem stated as follows: *Given an arbitrary window size N and an error bound ϵ, how to maintain ϵ-approximate variance of a stream of integers over sliding windows with size N in one pass?*

3.1.1.3 Our Contributions

In this research, we address the problem of maintaining ϵ-approximate variance of data streams over sliding windows [115]. To our knowledge, the best existing algorithm for this problem by Babcock et al. [17] requires $O(\frac{1}{\epsilon^2}\log N)$ space, and its running time is $O(1)$ amortized and $O(\frac{1}{\epsilon^2}\log N)$ in worst case. Datar et al. [35] presented a space lower bound

of $\Omega(\frac{1}{\epsilon}\log N)$ for maintaining the sum and hence the variance of the last N elements when assuming that the maximum absolute value of the integer elements is at most polynomial in N. Therefore, there is a gap of $\frac{1}{\epsilon}$ between the optimal bound and the best existing upper bound.

In this study, we propose the first ϵ-approximation algorithm to maintain variance of data streams over sliding windows that is optimal in both space and worst case time. Our algorithm requires $O(\frac{1}{\epsilon}\log N)$ space. Furthermore, its running time is $O(1)$ in worst case.

3.1.2 Algorithm

We first describe our algorithm, and then analyze its error bound, space requirement and running time.

3.1.2.1 Algorithm Description

Let N denote the window size and ϵ be the relative error parameter. An existing element is called *active* if it is one of the most recent N elements within the current window, or *expired* if it leaves the current window. For each element x_i, an index pos_i is used to record its position in the data stream, which is an indicator of "active" or "expired" by comparing with pos – the position index of the most recent element.

Similar to the algorithm in [17], elements in the data stream are partitioned into buckets. For each bucket B_i, we keep a timestamp t_i, which equals the position index of the oldest element in this bucket. Besides the timestamp, the following statistics information (n_i, μ_i, V_i) is also maintained:

n_i : number of elements in the bucket;

μ_i : mean of elements in the bucket;

V_i : variance of elements in the bucket.

Figure 3.1 shows the description of our algorithm. To estimate the variance of the data stream in current sliding window, we maintain a triplet $(n_{all}, \mu_{all}, V_{all})$ which objects are the

count, mean and variance of the combination of all active buckets in current window respectively. Note that n_{all}, μ_{all} and V_{all} are all initialized to zero at the beginning of the data stream.

Our algorithm has three steps when a new element arrives. When a new element x_t comes, the current index *pos* is updated. The new element constitutes a new bucket B_1 with $t_1 = pos$, $n_1 = 1$, $\mu_1 = x_t$, $V_1 = 0$, and an old bucket B_i becomes B_{i+1}. We update $(n_{all}, \mu_{all}, V_{all})$ by including the new bucket B_1 using Lemma 1.

Also, we check the oldest bucket B_m which has timestamp t_m. If it is expired, we first update $(n_{all}, \mu_{all}, V_{all})$ by excluding the expired bucket B_m using Lemma 1. Then we delete bucket B_m, which means our algorithm only maintains the buckets which elements are all active in the current window. We can use a wraparound counter with maximum $N - 1$ to represent the timestamps (i.e., position indices), since the oldest bucket will be deleted as soon as it expires. Therefore, the timing information of each bucket can be represented by $\lceil \log N \rceil$ bits.

We then check whether there are qualified pairs of adjacent buckets that can be merged to save space. Let $B_A = B_{i+1} \bigcup B_{i+2}$ and $B_B = \bigcup_{j=1}^{i} B_j$. We merge two adjacent buckets B_{i+1} and B_{i+2} if and only if they satisfy the following merging rules:

Rule 1. $V_{A,B} - V_B \leq \frac{\epsilon}{5} V_B$.

Rule 2. $n_A \leq \frac{\epsilon}{10} n_B$.

Rule 3. $n_A + n_B \leq \frac{N}{2}$.

We will explain why we set such merging rules when analyzing the error bound of our algorithm. When two adjacent buckets merge into a merged bucket, the merged bucket's timestamp is set to be the timestamp of the older one, and the merged bucket's statistics information can be calculated using Lemma 1. The merging step checks newer buckets first before checking older buckets.

Step 1: Insert New Element x_t.

Let $pos = pos + 1 \mod N$.
Create a new bucket for x_t. The new bucket becomes B_1 with $t_1 = pos$, $n_1 = 1$, $\mu_1 = x_t$, $V_1 = 0$.
(Note that an old bucket B_i becomes B_{i+1}.)
Update $(n_{all}, \mu_{all}, V_{all})$ by including new bucket B_1.

Step 2: Delete Expired Bucket.

Let B_m be the oldest bucket.
if there are at least 2 buckets and $t_m = pos$
  Update $(n_{all}, \mu_{all}, V_{all})$ by excluding bucket B_m.
  Delete bucket B_m.
endif

Step 3: Merge Buckets.

Let $i = 1$, and $B_B = B_1$.
while B_{i+2} exists
  Let $B_A = B_{i+1} \bigcup B_{i+2}$.
  if $n_A + n_B > \frac{N}{2}$
    return
  endif
  if $n_A \leq \frac{\epsilon}{10} n_B$ and $V_{A,B} - V_B \leq \frac{\epsilon}{5} V_B$
    Delete B_{i+2}, and let $B_{i+1} = B_A$.
    (Note that bucket B_{j+1} becomes B_j for $j \geq i + 2$.)
  else
    Let $i = i + 1$, and $B_B = B_B \bigcup B_i$.
  endif
endwhile

Figure 3.1 Algorithm Description

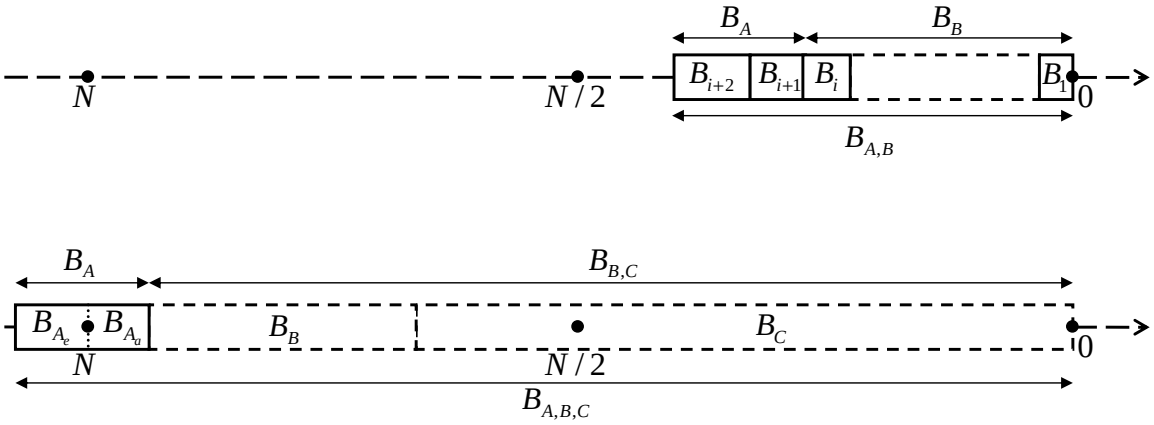


Figure 3.2 An Illustration of the Buckets

3.1.2.2 Error Bound

Before determining the space requirement and running time of our algorithm, we first prove that it can maintain ϵ-approximate variance of data streams over sliding windows. Lemma 1 gives the equations to calculate the statistics information of the combination of two buckets.

Lemma 1. *When two buckets B_i and B_j are merged into a single bucket $B_{i,j}$, the count, mean and variance of $B_{i,j}$ can be computed from those of B_i and B_j as follows:*

$$\begin{aligned} n_{i,j} &= n_i + n_j, \\ \mu_{i,j} &= \frac{n_i\mu_i + n_j\mu_j}{n_i + n_j}, \\ V_{i,j} &= V_i + V_j + \frac{n_i n_j}{n_i + n_j}(\mu_i - \mu_j)^2. \end{aligned} \tag{3.1}$$

Proof. This lemma and its proof can be found in [17]. □

Using Lemma 1, we can immediately get the following lemma which calculates the statistics information of the combination of three buckets.

Lemma 2. *When three buckets B_i, B_j and B_k are merged into a single bucket $B_{i,j,k}$, the count,*

mean and variance of $B_{i,j,k}$ can be computed as follows:

$$\begin{aligned} n_{i,j,k} &= n_i + n_j + n_k, \\ \mu_{i,j,k} &= \frac{n_i\mu_i + n_j\mu_j + n_k\mu_k}{n_i + n_j + n_k}, \\ V_{i,j,k} &= V_i + V_{j,k} + \frac{n_i(n_j + n_k)}{n_i + n_j + n_k}(\mu_i - \mu_{j,k})^2. \end{aligned}$$

Now let us analyze how estimation error is generated. For a bucket B_i which has $n_i = 1$, there is no estimation error when it expires, because either the single element or none element of B_i is active when the current window slides over B_i. Consequently, only the buckets generated by merging which have more than one element can introduce estimation error when expiring.

For instance, as shown in Figure 3.2, consider any merged bucket B_A which has more than one element, when it expires, it is split by the sliding window into two virtual buckets: the expired bucket B_{A_e} which contains all expired elements of B_A, and the active bucket B_{A_a} which contains all active elements of B_A. In Figure 3.2, we use doted boxes to denote the virtual buckets B_B and B_C. The virtual bucket B_B contains all newer elements when bucket B_A generates by merging, and the virtual bucket B_C contains all newer elements after B_B when bucket B_A expires. Note that B_C may become larger when more elements in B_A expires with the slide of the current window. Because we do not have the exact variance of bucket B_{A_a}, we have to return the estimated variance $\hat{V} = V_{B,C}$, which is the variance of all active buckets kept in memory space (i.e., V_{all}), to estimate the real variance.[1]

Let δ denote the relative error of the variance estimation of our algorithm. If the real variance $V = V_{A_a,B,C} = 0$, then according to Lemma 1 or 2, $\hat{V} = V_{B,C} \leq V_{A_a,B,C} = 0$, and $\delta = 0 < \epsilon$. When $V = V_{A_a,B,C} \neq 0$, δ is defined by

$$\delta = \frac{V - \hat{V}}{V} = \frac{V_{A_a,B,C} - V_{B,C}}{V_{A_a,B,C}}.$$

We will prove that $0 \leq \delta < \epsilon$ when $V = V_{A_a,B,C} \neq 0$ by utilizing the two virtual buckets B_B

[1] Although we may not know exact V_B and V_C since one merged bucket may expand the boundary between virtual buckets B_B and B_C, we can still calculate $V_{B,C}$ which is the variance of the combination of all active buckets.

and B_C. We first provide the relationship between n_B and n_C.

Lemma 3. *For any merged bucket B_A, let virtual bucket B_B contains all newer elements when bucket B_A generates by merging, and let virtual bucket B_C contains all newer elements after B_B when bucket B_A expires. We have*

$$n_B < n_C.$$

Proof. By construction, $n_{A_a} + n_B + n_C = N$. Thus, $n_C = N - (n_{A_a} + n_B)$. On the other hand, $n_{A_a} < n_A$, since n_{A_a} represents the number of active elements of the bucket A, and the expired portion of A is assumed to be non-empty. Thus, $n_C > N - (n_A + n_B)$. Now, by applying Rule 3 of merging rules (i.e., $n_A + n_B \leq \frac{N}{2}$), it follows that $N - (n_A + n_B) \geq \frac{N}{2}$. This implies that $n_C > \frac{N}{2}$. Since $n_{A_a} + n_B + n_C = N$, it results that $n_B < \frac{N}{2}$ since $n_{A_a} > 0$. Thus, $n_B < n_C$. □

To prove $0 \leq \delta < \epsilon$, we first relax the relative error δ to an auxiliary parameter θ, and then prove that $\theta < \epsilon$, where

$$\theta = \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2}.$$

Lemma 4. $0 \leq \delta \leq \theta$.

Proof. Equation (3.1) in Lemma 1 shows that the variance of a bucket is larger than or equal to that of its sub-bucket. We know that $B_{A,B,C} = B_{A_e} \bigcup B_{A_a,B,C} = B_A \bigcup B_{B,C}$, and thus $B_{B,C} \subset B_{A_a,B,C} \subset B_{A,B,C}$. Consequently, we get

$$V_{A,B,C} \geq V_{A_a,B,C} \geq V_{B,C}.$$

Therefore,

$$\begin{aligned} 0 \leq \delta &= \frac{V_{A_a,B,C} - V_{B,C}}{V_{A_a,B,C}} \leq \frac{V_{A,B,C} - V_{B,C}}{V_{B,C}} \\ &= \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + V_C + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2} \\ &\leq \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2} = \theta. \end{aligned}$$

□

Consequently, if we can prove that $\theta \leq \epsilon$, then our algorithm is an ϵ-approximation algorithm. We consider all possibilities of μ_C, which is the mean of virtual bucket B_C, in the following three cases separately. We will show that in each of these three cases, θ is less than ϵ. Without loss of generality, we assume that $\mu_A \leq \mu_B$. When $\mu_A > \mu_B$, we can make similar analysis and get the same conclusion, since actually we only utilize the order information between μ_A and μ_B.

Case 1. $\mu_B \leq \mu_C \leq 2\mu_B - \mu_A$

Lemma 5. *In Case 1, $\theta < \epsilon$.*

Proof. According to Rule 2, $n_A \leq \frac{\epsilon}{10} n_B < \frac{n_B}{10}$. Furthermore, Rule 1 shows that $V_{A,B} - V_B \leq \frac{\epsilon}{5} V_B$. We only consider that $V_{A,B} \neq 0$, thereby $V_B \neq 0$. Thus, we get

$$\begin{aligned} \epsilon &\geq 5\frac{V_{A,B} - V_B}{V_B} = 5\frac{V_A + \frac{n_A n_B}{n_A+n_B}(\mu_A - \mu_B)^2}{V_B} \\ &\geq 5\frac{V_A + \frac{n_A n_B}{0.1 n_B+n_B}(\mu_A - \mu_B)^2}{V_B} \\ &= 5\frac{V_A + \frac{n_A}{1.1}(\mu_A - \mu_B)^2}{V_B}. \end{aligned}$$

Therefore,

$$\epsilon > 4\frac{V_A + n_A(\mu_A - \mu_B)^2}{V_B}. \tag{3.2}$$

Because $\mu_B \leq \mu_C$ in Case 1, and the mean of a merged bucket should be bounded between the means of its two sub-buckets, thus $\mu_B \leq \mu_{B,C} \leq \mu_C$. Therefore, in Case 1 that $\mu_B \leq \mu_C \leq$

$2\mu_B - \mu_A$, we get

$$\mu_B \leq \mu_{B,C} \leq 2\mu_B - \mu_A.$$

We have assumed that $\mu_A \leq \mu_B$, therefore

$$0 \leq \mu_B - \mu_A \leq \mu_{B,C} - \mu_A \leq 2(\mu_B - \mu_A).$$

We get $|\mu_A - \mu_{B,C}| \leq 2|\mu_A - \mu_B|$. Consequently,

$$\begin{aligned}
\theta &= \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2} \\
&\leq \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B} \\
&\leq \frac{V_A + n_A\frac{n_B+n_C}{n_A+n_B+n_C}4(\mu_A - \mu_B)^2}{V_B} \\
&\leq \frac{V_A + 4n_A(\mu_A - \mu_B)^2}{V_B} \\
&\leq 4\frac{V_A + n_A(\mu_A - \mu_B)^2}{V_B}.
\end{aligned}$$

Compared with inequality (3.2), we get $\theta < \epsilon$. □

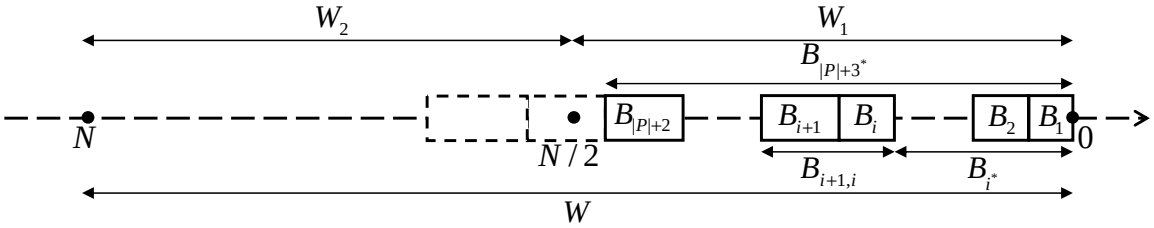


Figure 3.3 An Illustration of the Buckets and Windows

Case 2. $\mu_C > 2\mu_B - \mu_A$

Lemma 6. *In Case 2, $\theta < \epsilon$.*

Proof. In this case that $\mu_C > 2\mu_B - \mu_A$, we get

$$\mu_B - \mu_A < \mu_C - \mu_B.$$

We have assumed that $\mu_A \leq \mu_B$, therefore $\mu_B < \mu_C$. Also, the mean of a merged bucket should be bounded between the means of its two sub-buckets, therefore $\mu_B < \mu_{B,C} < \mu_C$. Consequently,

$$\begin{aligned} |\mu_A - \mu_{B,C}| &< \mu_C - \mu_A = \mu_C - \mu_B + \mu_B - \mu_A \\ &< 2|\mu_B - \mu_C|. \end{aligned} \tag{3.3}$$

Therefore,

$$\begin{aligned} \theta &= \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2} \\ &= \frac{V_A}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2} + \frac{\frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2} \\ &< \frac{V_A}{V_B} + \frac{\frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{\frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2} \\ &= \frac{V_A}{V_B} + \frac{n_A}{n_B} \cdot \frac{(n_B+n_C)^2}{(n_A+n_B+n_C)n_C} \cdot (\frac{|\mu_A - \mu_{B,C}|}{|\mu_B - \mu_C|})^2. \end{aligned}$$

Equation (3.1) in Lemma 1 shows that the variance of a bucket is larger than or equal to the sum of variance of both sub-buckets. Therefore $V_A \leq V_{A,B} - V_B$. According to Rule 1, we get

$$\frac{V_A}{V_B} \leq \frac{V_{A,B} - V_B}{V_B} \leq \frac{\epsilon}{5}.$$

According to Rule 2, we get

$$\frac{n_A}{n_B} \leq \frac{\epsilon}{10}.$$

From inequality (3.3), we get

$$(\frac{|\mu_A - \mu_{B,C}|}{|\mu_B - \mu_C|})^2 < 4.$$

Let $g(n_C) = \frac{(n_B+n_C)^2}{(n_A+n_B+n_C)n_C}$. Consequently,

$$\theta < \frac{\epsilon}{5} + \frac{\epsilon}{10} \cdot g(n_C) \cdot 4 = \frac{\epsilon}{5}(1 + 2g(n_C)).$$

The derivative of $g(n_C)$ is

$$\begin{aligned} g'(n_C) &= \frac{(n_B+n_C)[(n_A-n_B)n_C-(n_A+n_B)n_B]}{(n_A+n_B+n_C)^2 n_C^2} \\ &< -\frac{(n_A+n_B)(n_B+n_C)n_B}{(n_A+n_B+n_C)^2 n_C^2} < 0. \end{aligned}$$

Therefore, $g(n_C)$ decreases with the increase of n_C. According to Lemma 3 that $n_B < n_C$,

$$g(n_C) < g(n_B) = \frac{4n_B}{n_A+2n_B} < \frac{4n_B}{2n_B} = 2.$$

Therefore,

$$\theta < \frac{\epsilon}{5}(1+2g(n_C)) < \epsilon.$$

□

Case 3. $\mu_C < \mu_B$

Lemma 7. *In Case 3,* $\theta < \epsilon$.

Proof. We claim that for any instance B_C in Case 3, we can find an equal or even worse instance $B_{C'}$ in terms of θ which mean belongs to either Case 1 or Case 2, such that $\theta < \theta'$ where

$$\theta' = \frac{V_A + \frac{n_A(n_B+n_{C'})}{n_A+n_B+n_{C'}}(\mu_A-\mu_{B,C'})^2}{V_B + \frac{n_B n_{C'}}{n_B+n_{C'}}(\mu_B-\mu_{C'})^2}.$$

If $\mu_C < \mu_B$, we construct $B_{C'}$ from B_C by adding $2(\mu_B-\mu_C)$ to each element in B_C. For bucket $B_{C'}$,

$$\begin{aligned} n_{C'} &= n_C, \\ \mu_{C'} &= 2\mu_B - \mu_C > \mu_B. \end{aligned}$$

Consequently, the mean $\mu_{C'}$ of bucket $B_{C'}$ belongs to either Case 1 or Case 2, and thus $\theta' < \epsilon$ according to Lemma 5 and 6. Since μ_C and $\mu_{C'}$ are symmetric about μ_B, $\mu_{B,C}$ and $\mu_{B,C'}$

should also be symmetric about μ_B. Therefore $\mu_{B,C} + \mu_{B,C'} = 2\mu_B$, and

$$\mu_{B,C'} - \mu_B = \mu_B - \mu_{B,C} > 0.$$

We have assumed that $\mu_A \leq \mu_B$, therefore

$$\begin{aligned} |\mu_A - \mu_{B,C}| &= |(\mu_B - \mu_{B,C}) - (\mu_B - \mu_A)| \\ &= |(\mu_{B,C'} - \mu_B) - (\mu_B - \mu_A)| \\ &\leq |(\mu_{B,C'} - \mu_B) + (\mu_B - \mu_A)| \\ &= |\mu_A - \mu_{B,C'}|, \end{aligned}$$

and

$$\begin{aligned} \theta &= \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2} \\ &= \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_{C'})^2} \\ &\leq \frac{V_A + \frac{n_A(n_B+n_{C'})}{n_A+n_B+n_{C'}}(\mu_A - \mu_{B,C'})^2}{V_B + \frac{n_B n_{C'}}{n_B+n_{C'}}(\mu_B - \mu_{C'})^2} \\ &= \theta'. \end{aligned}$$

Consequently, we get $\theta \leq \theta' < \epsilon$ in Case 3. □

We have shown that, when the real variance $V = 0$, our algorithm returns the estimation $\hat{V} = 0$. When the real variance $V \neq 0$, according to Lemma 4, 5, 6 and 7, we get $0 \leq \delta \leq \theta < \epsilon$ where δ denotes the relative error of the variance estimation of our algorithm. Consequently, we get the following theorem about the error bound of our algorithm.

Theorem 1. *Our algorithm can maintain ϵ-approximate variance of data streams over sliding windows.*

Furthermore, we make the observation that our algorithm only makes one-sided error. That

is,

$$(1-\epsilon)V \leq \hat{V} \leq V.$$

To help understand our algorithm, we shortly explain why we set such three merging rules and why these three merging rules can bound the estimation error to $O(\epsilon)$. As shown in Lemma 4,

$$0 \leq \delta \leq \theta = \frac{V_A + \frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2}.$$

First of all, Rule 1 implies that $O(\frac{V_A}{V_B}) = O(\epsilon)$. Furthermore, when μ_C is close to μ_B (i.e., Case 1), Rule 1 and 2 can guarantee that

$$O(\frac{\frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B}) \approx O(\epsilon).$$

However, when μ_C is far way from μ_A and μ_B (i.e., Case 2), in worst case

$$\begin{aligned} O(\frac{\frac{n_A(n_B+n_C)}{n_A+n_B+n_C}(\mu_A - \mu_{B,C})^2}{V_B + \frac{n_B n_C}{n_B+n_C}(\mu_B - \mu_C)^2}) &\approx O(\frac{n_A}{n_B} \cdot \frac{(n_B+n_C)^2}{(n_A+n_B+n_C)n_C}) \\ &\approx O(\epsilon \cdot \frac{N}{n_C}). \end{aligned}$$

Therefore we have to add a constraint on n_C (i.e., Rule 3), such that $O(n_C) = O(N)$ to bound relative error to $O(\epsilon)$.

3.1.2.3 Space Requirement

Now consider any two adjacent buckets B_i and B_{i+1} when $i \geq 2$ after the merging procedure completes. Let bucket $B_{i+1,i} = B_{i+1} \bigcup B_i$ and bucket $B_{i^*} = \bigcup_{j=1}^{i-1} B_j$, and Figure 3.3 shows an illustration. When $n_{i+1,i} + n_{i^*} = n_{i+2^*} \leq \frac{N}{2}$, according to the merging rules, our algorithm has the following invariant:

Invariant 1. *For $i \geq 2$, if B_{i+1} exists, $B_{i+1,i}$ has either*

$$\textit{Property 1: } n_{i+1,i} = n_{i+2^*} - n_{i^*} > \frac{\epsilon}{10} n_{i^*}, \textit{ or}$$
$$\textit{Property 2: } V_{i+1,i,i^*} - V_{i^*} = V_{i+2^*} - V_{i^*} > \frac{\epsilon}{5} V_{i^*}.$$

Let W denote the current N-sized window, W_1 denote the current $\frac{N}{2}$-sized window, and W_2 denote the past $\frac{N}{2}$-sized window. We define three sets, P, P_1 and P_2. P is all adjacent bucket pairs in W_1 except the first pair (i.e. $B_1 \bigcup B_2$). P_1 is a subset of P which contains all adjacent bucket pairs with Property 1, and P_2 is a subset of P which contains all adjacent bucket pairs with Property 2.

$$\begin{aligned} P &= \{B_{i+1,i} : n_{i+2^*} \leq \frac{N}{2}, i \geq 2\}, \\ P_1 &= \{B_{i+1,i} : n_{i+2^*} > (1 + \frac{\epsilon}{10}) n_{i^*}, B_{i+1,i} \in P\}, \\ P_2 &= \{B_{i+1,i} : V_{i+2^*} > (1 + \frac{\epsilon}{5}) V_{i^*}, B_{i+1,i} \in P\}. \end{aligned}$$

Then from Invariant 1, we get

$$\begin{aligned} P &= P_1 \bigcup P_2, \\ |P| &\leq |P_1| + |P_2|. \end{aligned}$$

Let m denote the number of buckets within the current N-sized window W. Let m_1 denote the number of buckets within the current $\frac{N}{2}$-sized window W_1. Let m_2 denote the number of buckets within the past $\frac{N}{2}$-sized window W_2. Then

$$m_1 = |P| + 2.$$

Consider any window W_2, all buckets in it are from a particular old W_1 after potential merging. According to Rule 3, any bucket which passes the boundary between W_1 and W_2 will not change

any more in W_2 before it expires. Consequently, at any time,

$$m_2 \leq \max(m_1).$$

In addition, there is at most one bucket which expands the boundary between W_1 and W_2. Consequently,

$$m \leq m_1 + m_2 + 1 \leq 2\max(m_1) + 1 = 2\max(|P|) + 5. \tag{3.4}$$

Therefore, if we can find the upper bound of $|P|$, we get the bound of m (i.e., the number of buckets in our algorithm) accordingly.

Lemma 8. *In the case that every $B_{i+1,i}$ in P has Property 1 (i.e., $P = P_1$), we have*

$$|P_1| < 2\lceil\frac{10}{\epsilon}\rceil \log N - 1.$$

Proof. In this case, since any adjacent bucket pair $B_{i+1,i}$ has Property 1 when $i \geq 2$, then for any index $i + 2j \leq |P_1| + 3$ with $i \geq 2$,

$$\begin{aligned}
n_{i+2^*} &> (1 + \frac{1}{10}\epsilon)n_{i^*}, \\
n_{i+4^*} &> (1 + \frac{1}{10}\epsilon)n_{i+2^*} > (1 + \frac{2}{10}\epsilon)n_{i^*}, \\
n_{i+6^*} &> (1 + \frac{1}{10}\epsilon)n_{i+4^*} > (1 + \frac{3}{10}\epsilon)n_{i^*}, \\
&\ldots \\
n_{i+2j^*} &> (1 + \frac{j}{10}\epsilon)n_{i^*}.
\end{aligned}$$

Thus when $j = \lceil\frac{10}{\epsilon}\rceil$, $n_{i+2j^*} > 2n_{i^*}$. Consequently, n_{i^*} will be doubled when i increases every $2\lceil\frac{10}{\epsilon}\rceil$. We know that $n_{2^*} = n_1 = 1$, therefore

$$\begin{aligned}
2^{\lfloor\frac{|P_1|+1}{2\lceil\frac{10}{\epsilon}\rceil}\rfloor} \cdot n_{2^*} &\leq n_{|P_1|+3^*} \leq \frac{N}{2}, \\
2^{\frac{|P_1|+1}{2\lceil\frac{10}{\epsilon}\rceil}} &< N, \\
|P_1| &< 2\lceil\frac{10}{\epsilon}\rceil \log N - 1.
\end{aligned}$$

□

Let R be the upper bound on the absolute value of the data elements.

Lemma 9. *In the case that every $B_{i+1,i}$ in P has Property 2 (i.e., $P = P_2$), we have*

$$|P_2| < 2\lceil\frac{5}{\epsilon}\rceil \log(\frac{3NR^2}{2}) + 1.$$

Proof. The proof is similar to that of Lemma 8. In this case, the maximum element number bound $\frac{N}{2}$ in window W_1 becomes the maximum variance bound $\frac{N}{2}R^2$. V_{i^*} will be doubled when i increases every $2\lceil\frac{5}{\epsilon}\rceil$.

We claim that $V_{4^*} \geq \frac{2}{3}$. First, $V_{4^*} > 0$, otherwise $V_{4^*} = V_{2^*} = 0$ which violates Property 2. Because $V_{4^*} > 0$, and all elements in the data stream are integers, then V_{4^*} achieves its minimum value $\frac{2}{3}$ if and only if the three elements in B_{4^*} are $\{c_1, c_1, c_1 \pm 1\}$, where c_1 is an integer. Consequently,

$$\begin{aligned} 2^{\lfloor\frac{|P_2|-1}{2\lceil\frac{5}{\epsilon}\rceil}\rfloor} \cdot V_{4^*} &\leq V_{|P_2|+3^*} \leq \frac{NR^2}{2}, \\ 2^{\frac{|P_2|-1}{2\lceil\frac{5}{\epsilon}\rceil}} &< \frac{3NR^2}{2}, \\ |P_2| &< 2\lceil\frac{5}{\epsilon}\rceil \log(\frac{3NR^2}{2}) + 1. \end{aligned}$$

□

Lemma 10. *In any case, we have*

$$|P| < 2\lceil\frac{10}{\epsilon}\rceil \log N + 2\lceil\frac{5}{\epsilon}\rceil \log(\frac{3NR^2}{2}).$$

Proof. We check the size of subsets P_1 and P_2 separately. First, consider how many pairs of adjacent buckets that have Property 1 (belong to P_1) can exist to contain at most $\frac{N}{2}$ elements. Any pair of adjacent buckets which does not belong to P_1 also contributes to contain some number of elements. Further, n_{i^*} is still doubled after every $2\lceil\frac{10}{\epsilon}\rceil$ pairs of adjacent buckets in P_1. Consequently, Lemma 8 is still true for $|P_1|$. Similarly, any pair of adjacent buckets which

does not belong to P_2 also contributes to the increase of variance, and Lemma 9 is still true for $|P_2|$. Therefore,

$$|P| \leq |P_1| + |P_2| < 2\lceil\frac{10}{\epsilon}\rceil \log N + 2\lceil\frac{5}{\epsilon}\rceil \log(\frac{3NR^2}{2}).$$

□

Consequently, we get the space bound of our algorithm as shown in the following theorem.

Theorem 2. *Our algorithm can maintain ϵ-approximate variance of data streams over sliding windows using $O(\frac{1}{\epsilon}\log N)$ space.*

Proof. According to inequality (3.4) and Lemma 10, we get

$$\begin{aligned} m &\leq 2\max(|P|) + 5 \\ &< 4\lceil\frac{10}{\epsilon}\rceil\lceil\log N\rceil + 4\lceil\frac{5}{\epsilon}\rceil\lceil\log(\frac{3NR^2}{2})\rceil + 5. \end{aligned}$$

Therefore, our algorithm is $O(\frac{1}{\epsilon}\log(NR^2))$ in space. Assume that R is at most polynomial in N, the space requirement is in fact $O(\frac{1}{\epsilon}\log N)$. □

3.1.2.4 Running Time

Obviously, the running time of our algorithm is $O(\frac{1}{\epsilon}\log N)$ in worst case. Although Step 1 and 2 are both $O(1)$, Step 3 needs $O(\frac{1}{\epsilon}\log N)$ in worst case. Now we propose a mechanism which makes our algorithm an $O(1)$ running time algorithm in worst case. The baseline idea is that we only check constant number of pairs of adjacent buckets each time in the third (merging) step.

Let c be an integer which is large than 1. When each element arrives, we still insert it as a new bucket (Step 1) and delete expired bucket if any (Step 2). However, instead of checking all pairs of adjacent buckets within W_1 in Step 3, we only run the kernel **while** loop c times. After that, the merging procedure is hanged up and will be resumed with the same status when a new element arrives. Note that before the merging procedure completes, each new element

is stored in an individual bucket, and window W_1 does not slide (although window W slides and the expired buckets are still deleted). Since many new buckets may be queued before the merging procedure completes, we need extra space to keep them. However, such extra space can still be efficiently bounded.

Lemma 11. *Let Y denote the maximum number of buckets that window W_1 can have after merging procedure completes which is in $O(\frac{1}{\epsilon}\log N)$ as shown in Theorem 2. Let Z denote the maximum number of new arriving elements (buckets) during the merging procedure. Then*

$$Z \leq \lceil \frac{Y}{c-1} \rceil.$$

Proof. Let y_i denote the number of buckets that window W_1 can have after the ith merging procedure completes. Let z_i denote the number of new arriving elements during the ith merging procedure. Then for the $i+1$th merging procedure, in the beginning, window W_1 can contain at most $y_i + z_i$ buckets and at most $y_i + z_i - 2$ pairs of adjacent buckets which need to check after the first bucket B_1. Because we check c pairs of adjacent buckets when receiving a new element, we have

$$z_{i+1} \leq \lceil \frac{y_i + z_i - 2}{c} \rceil.$$

We claim that for any index $i \geq 1$, $z_i \leq \lceil \frac{Y}{c-1} \rceil$. We use induction to prove this claim. First, $z_1 = 1$. Suppose for an index $i > 1$, the claim is true. Then for index $i+1$,

$$\begin{aligned} z_{i+1} &\leq \lceil \frac{y_i + z_i - 2}{c} \rceil \leq \lceil \frac{Y + \lceil \frac{Y}{c-1} \rceil - 2}{c} \rceil \\ &\leq \lceil \frac{Y + \frac{Y}{c-1}}{c} \rceil \leq \lceil \frac{Y}{c-1} \rceil. \end{aligned}$$

Therefore, we get $Z \leq \lceil \frac{Y}{c-1} \rceil$. □

Lemma 11 shows that we can process a new element using constant time while the extra space and thus the total space are still in $O(\frac{1}{\epsilon}\log N)$. Further, it is clear that our algorithm can update $(n_{all}, \mu_{all}, V_{all})$ in $O(1)$, therefore our algorithm can answer a query of the variance in

the current sliding window with constant time. Consequently, we get the final theorem which concludes the optimal properties of our algorithm:

Theorem 3. *Our algorithm can maintain ϵ-approximate variance of data streams over sliding windows using $O(\frac{1}{\epsilon}\log N)$ space. Furthermore, the running time of processing a new element and answering a query is $O(1)$ in worst case. Our algorithm is optimal in both space and worst case time.*

3.1.3 Conclusions

In this research, we address the problem of maintaining ϵ-approximate variance in data streams over sliding windows. Our proposed algorithm only needs $O(\frac{1}{\epsilon}\log N)$ space and $O(1)$ running time in worst case. Therefore, it is optimal in both space and worst case time. In the future, we will continue to extend our algorithm to other data stream statistics problems over sliding windows.

3.2 Frequency Estimation over Sliding Windows

3.2.1 Introduction

The problem of computing characteristics of large data streams has received considerable attention [18, 82]. Many characteristics of large data streams, such as frequency, quantile, sum, mean, variance, diameter, top-k list (hot list), distribution, etc., have been widely studied. If we have sufficient large space and do not have time constraints, we can precisely obtain any characteristics that we want. However, the issue in processing large data streams is that in many cases we probably only have one chance to process each item in the data streams. We cannot store all data because of constraints in memory space or privacy issue. Therefore, in these cases we have to gather the interested characteristics with only one pass.

An even greater challenge is to process data streams over sliding windows. An algorithm which works over sliding windows not only can gather the data streams' characteristics, but also update the characteristics by inserting new data and deleting expired data. Unfortunately, many previous data stream algorithms cannot work over sliding windows.

3.2.1.1 Motivation

Frequency is a fundamental characteristic in many data mining applications. For instance, it can be used in sensor data mining, analysis of web query logs, network measurement and monitoring, bandwidth statistics for billing purposes, transaction analysis in stocks, and iceberg queries [44] in large-scale data bases, etc.

In recent years, such problems are considered under sliding window models. The advantage of an algorithm which works over sliding windows is that it can get rid of the stale data and only consider the fresh data, which is meaningful in many cases. For instance, in an intrusion detection system (IDS), the current status of the network is usually more important than that of one day ago. The characteristics gathered over sliding windows can provide a more smooth view of the data stream. However, many previous algorithms cannot work over sliding windows. Recently, several data mining algorithms over sliding windows are proposed [16, 17, 35, 45, 50, 51, 67, 104, 115, 121]. For the problem of ϵ-approximate frequency estimation over sliding windows, Arasu and Manku [16] presented an algorithm which requires $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$ space and $O(\log\frac{1}{\epsilon})$ running time. Recently, an algorithm proposed by Lee and Ting [65] achieves $O(\frac{1}{\epsilon})$ space, but needs $O(\frac{1}{\epsilon})$ processing time for update and query. There is no algorithm which can achieve both linear space in terms of $\frac{1}{\epsilon}$ and constant running time simultaneously. In this research, we are trying to design an efficient frequency estimation algorithm which needs $O(\frac{1}{\epsilon})$ space and $O(1)$ running time.

3.2.1.2 Problem Definition

ϵ-Approximate Frequency Estimation Suppose we have a data stream with N items, and each item is in the set I. A frequency estimation algorithm is an **ϵ-approximate** algorithm if it guarantees that for any item $i \in I$, the error between true frequency f_i and estimated frequency $\hat{f}_i$ is bounded by

$$0 \leq f_i - \hat{f}_i \leq \epsilon N. \tag{3.5}$$

Sliding Window There are two common types of sliding windows, *count-based* windows which maintain the last (most recent) N items in the stream, and *time-based* windows which maintain all items that arrived in the last T time units. Therefore, the time span of a count-based window may vary, while the number of items in a time-based window may change from time to time. For example, let $< i, t >$ denote that an item i arrives at timestamp t. We have a stream of items as

$$< i_1, 1 >, < i_2, 2 >, < i_3, 5 >, < i_1, 6 >, < i_4, 7 >, \ldots$$

If the sliding window is count-based with $N = 3$, then the item sets in the sliding windows follow $\{i_1\}$, $\{i_1, i_2\}$, $\{i_1, i_2, i_3\}$, $\{i_2, i_3, i_1\}$, $\{i_3, i_1, i_4\}$, ... If the sliding window is time-based with $T = 2.1$, then the item sets in the sliding windows follow $\{i_1\}$, $\{i_1, i_2\}$, $\{i_2\}$, $\{\phi\}$, $\{i_3\}$, $\{i_3, i_1\}$, $\{i_3, i_1, i_4\}$, ...

Problem Statement In this research, we consider the problem stated as follows: *Given an arbitrary window size N and an error bound ϵ, how to maintain ϵ-approximate frequency estimation of a data stream over count-based sliding windows with size N in one pass?*

We leave the problem of frequency estimation over time-based sliding windows to our future work.

3.2.1.3 Our Contributions

To our knowledge, the best existing algorithms for the problem of estimating ϵ-approximate frequency in data streams over sliding windows either require $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$ space and $O(\log\frac{1}{\epsilon})$ running time [16], or require $O(\frac{1}{\epsilon})$ both in space and running time [65]. Our contribution in this research is that we propose two novel deterministic algorithms, Snapshot-Basic and Snapshot-Advanced, which can maintain ϵ-approximate frequency estimation over sliding windows only using $O(\frac{1}{\epsilon})$ space [117]. Furthermore, Snapshot-Advanced is the first efficient deterministic algorithm which can achieve $O(\frac{1}{\epsilon})$ space requirement and only needs $O(1)$ running time to process each item in the data stream and to answer a query. In addition, as an

application of our algorithms, we extend them to solve the problem of estimating flow size.

3.2.2 SNAPSHOT Algorithms

Let N denote the window size and ϵ be the error parameter. To simplify the description, we assume that $\frac{1}{\epsilon}$ is an integer, and ϵN can be divided by 3. We first propose a basic algorithm named Snapshot-Basic, which is straightforward but needs quite a number of operations to process an arrival item. Then we introduce a more sophisticated algorithm named Snapshot-Advanced which can work in $O(1)$ running time.

3.2.2.1 Basic Algorithm

The baseline idea of Snapshot-Basic is to snapshoot items' position information so that we can use the snapshots to update estimates and bound the estimation error over sliding windows which is similar to the idea in [65]. For instance, we take snapshots on the positions of the 1st, 101st, 201st, $\cdots$ identical items. When a snapshot expires, we can decrease the corresponding item's frequency estimates by 100, therefore it is possible to bound the estimation error to the level of 100.

Figure 3.4 describes the Snapshot-Basic algorithm. A linked list L is kept with at most $\frac{3}{\epsilon}$ item entries, which is called *item list*. In each entry of the item list, we keep an item identifier, a counter $\hat{f}$ as its frequency estimate, and a pointer to another linked list which stores the snapshots of this item. It means that each item entry has its own *snapshot list*. When a new item i arrives, and the item list L is not full, we create an entry for it in L. We set its frequency estimate $\hat{f}_i$ to 1. We also snapshoot the current position, which means that we add the current position index n into this item's snapshot list. Instead of keeping all its positions, we only snapshoot the 1st, $(\frac{\epsilon N}{3}+1)$th, $(\frac{2\epsilon N}{3}+1)$th, $(\epsilon N+1)$th, $\cdots$ positions of item i. When an old item arrives, we simply increase its frequency estimate by 1, and snapshoot if necessary. In the case that the arrival item i is a new item, and L has kept $\frac{3}{\epsilon}$ distinct items, we decrease the frequency estimate of each item by 1. We delete all snapshots and items which are no need to keep. To update the current sliding window, when an item arrives, we first find the oldest

snapshot among all snapshots of all items. If this oldest snapshot is out of the current window, this snapshot is deleted, and the corresponding item's frequency estimate is decreased by $\frac{\epsilon N}{3}$. We delete this item from the item list L if its frequency estimate is less than or equal to 0.

Step 1: Delete expired snapshot and item.

Find the oldest snapshot.
If it is expired and belongs to item j, delete this snapshot, and decrease $\hat{f}_j$ by $\frac{\epsilon N}{3}$.
If $\hat{f}_j \leq 0$, delete item j from list L.

Step 2: Process the arrival item i.

Case 1: i is an old item (i.e., $i \in L$).

Increase $\hat{f}_i$ by 1.
If $\hat{f}_i = 1 \bmod \frac{\epsilon N}{3}$, snapshoot the current position.

Case 2: i is a new item, and L is not full.

Create an entry for item i in list L.
Set $\hat{f}_i$ to 1 and snapshoot the current position.

Case 3: i is a new item, and L is full of $\frac{3}{\epsilon}$ items.

For each item j in list L:
  Decrease $\hat{f}_j$ by 1.
  If $\hat{f}_j = 0 \bmod \frac{\epsilon N}{3}$, delete the most recent snapshot
    from j's snapshot list.
  If $\hat{f}_j = 0$, delete item j from L.

Figure 3.4 SNAPSHOT-BASIC Algorithm Description

Figure 3.6 shows an example of how SNAPSHOT-BASIC works. In this example, $N = 18$, $\epsilon = \frac{1}{2}$, and the input data stream is shown in Figure 3.5. The item list will keep at most 6 ($= \frac{3}{\epsilon}$) entries, and positions are snapshot for each 3 ($= \frac{\epsilon N}{3}$) identical items. The up arrow "↑" in Figure 3.5 shows that there is a snapshot taken on that position. Figure 3.6(a) shows the item list L when $n = 18$, and there are 6 items in list L and totally 8 snapshots in all snapshot lists, so there is no room for new items in list L. When the 19th item i_3 arrives, the snapshot $s_{i_1,1}$ expires and is deleted, and $\hat{f}_{i_1}$ is decreased by 3. Also, a new snapshot entry $s_{i_3,2}$ is inserted into the head of i_3's snapshot list as shown in Figure 3.6(b). When the 20th item i_7 arrives, no snapshot expires. Because i_7 is not present in the item list L and there is no room to create a new item entry for i_7, all counters in the item list are decreased by 1. The

position	1	2	3	4	5	6	7	8	9	10	11	12
item	i_1	i_1	i_2	i_3	i_3	i_4	i_5	i_1	i_4	i_6	i_2	i_5
snapshots in SNAPSHOT-BASIC	$\uparrow$		$\uparrow$	$\uparrow$		$\uparrow$	$\uparrow$			$\uparrow$		
snapshots in SNAPSHOT-ADVANCED	$\uparrow_c$		$\uparrow_c$	$\uparrow_c$		$\uparrow_p$	$\uparrow_c$			$\uparrow_p$		

position	13	14	15	16	17	18	19	20	21	22	23	$\cdots$
item	i_2	i_3	i_1	i_6	i_5	i_2	i_3	i_7	i_8	i_9	i_3	$\cdots$
snapshots in SNAPSHOT-BASIC			$\uparrow$			$\uparrow$	$\uparrow$	$\Downarrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\cdots$
snapshots in SNAPSHOT-ADVANCED			$\uparrow_p$			$\uparrow_p$	$\uparrow_p$	$\uparrow_p$	$\Downarrow$	$\uparrow_p$	$\uparrow$	$\cdots$

Figure 3.5 An Example Data Stream

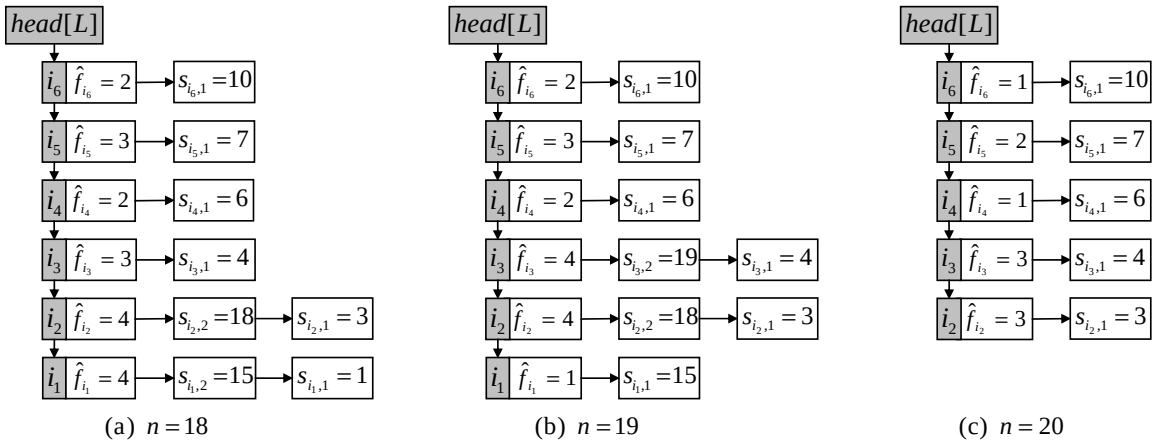


Figure 3.6 Example Item List when Window Slides in SNAPSHOT-BASIC.

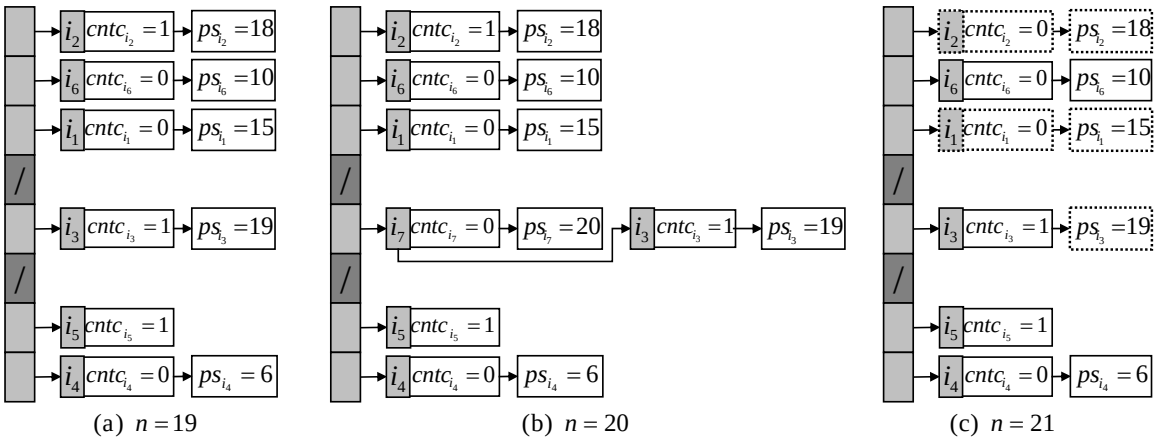


Figure 3.7 Example Hash Table when Window Slides in SNAPSHOT-ADVANCED.

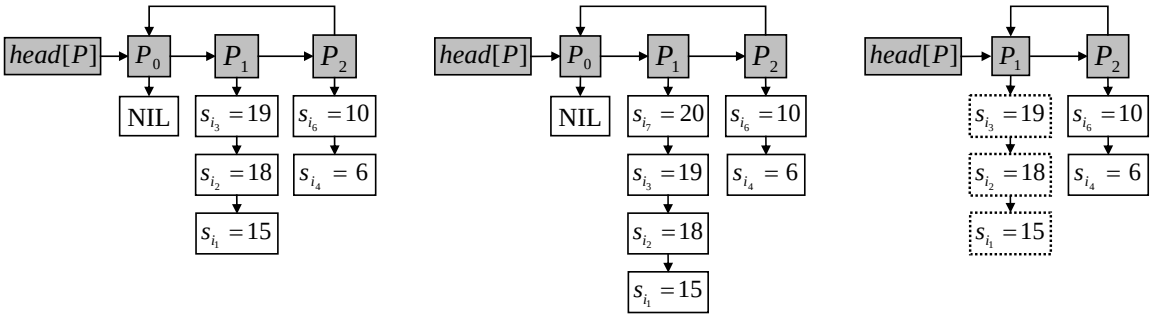


Figure 3.8 Example Partial Snapshot List when Window Slides in SNAPSHOT-ADVANCED.

down arrow "⇓" shows that there is a *decrease operation* in this position. i_1 is deleted from list L as its frequency estimate is equal to 0. Also, snapshots $s_{i_2,2}$ and $s_{i_3,2}$ are deleted from i_2's and i_3's snapshot lists as shown in Figure 3.6(c).

It is possible that an inserted item is deleted from L and appears again in the data stream after the deletion. In this case, it is treated as a new item because there is no entry (memory) for it in L. For example, when we snapshoot the 5th i_3 on position 23, the old entry and snapshots for i_3 have been expired or deleted, so i_3 is reinserted into the item list L again as a new item. Also, when we snapshoot the $(\frac{x\epsilon N}{3}+1)$th position of an item, it is possible that this position is actually the $(\frac{x\epsilon N}{3}+1+y)$th position of this item in the data stream if the decrease operations in Case 3 of Step 2 have happened y times, where x and y are non-negative integers.

Now we prove the correctness of the SNAPSHOT-BASIC algorithm, and give the space requirement.

Theorem 4. *For any item i, no matter whether it appears in the data stream or not,* SNAPSHOT-BASIC *can maintain ϵ-approximate frequency estimation over sliding windows with size at most N. Furthermore,* SNAPSHOT-BASIC *uses $O(\frac{1}{\epsilon})$ space.*

Proof. Approximation:

The error of frequency estimation comes from two sources. One is the operation to clear the expired snapshot in Step 1, and the other is the decrease operation in Case 3 of Step 2. In Step 1, once a snapshot just expires, we should only decrease the corresponding item's frequency

estimate by 1. However, SNAPSHOT-BASIC decreases it by $\frac{\epsilon N}{3}$. Then this operation will introduce a negative error at most $\frac{\epsilon N}{3} - 1$. Fortunately, this kind of error will not accumulate for the same item during data stream when multiple snapshots of the same item expire and are deleted. The reason is that, when the next snapshot of the same item expires, the sliding window has moved on and the error introduced by the previous expired snapshot is out of the current window and decreases to zero. Therefore, the error introduced by the operation in Step 1 can be bounded to $\frac{\epsilon N}{3} - 1$ at any time for any items.

When Case 3 of Step 2 happens, all existing items must decrease their frequency estimates by 1, and the new item cannot be inserted. Therefore, this operation will introduce a negative error equal to 1 for certain items. Now we calculate how many times this decrease operation can be performed in any current window not wider than N. Suppose the current window size is N'.[2] Let the last(most recent) item's position index in the current window be n. So the items in the current window is $\{e_{n-N'+1}, e_{n-N'+2}, \ldots, e_n\}$. Let sum_j denote the sum of all frequency estimates in the item list L when the current item's position index is j. So $sum_{n-N'}$ is the sum of all frequency estimates in the item list just before entering the current window, and sum_n is the sum of all frequency estimates in the item list at the end of the current window. There are N' new arrival items in the current window, so the maximum value that can be decreased during the current window is $sum_{n-N'} - sum_n + N'$. The operations in Step 1 guarantee that no expired item can be counted, so $sum_{n-N'} \leq N$. In the worst case, $sum_n = 0$, $sum_{n-N'} = N$, and Step 1 is not performed (which means that all frequency estimate decreases are from Case 3 of Step 2). Each time the decrease operation decreases the sum exactly by $\frac{3}{\epsilon}$ plus 1[3]. Therefore, the number of the decrease operations in Case 3 of Step 2 is at most

$$\frac{sum_{n-N'} - sum_n + N'}{\frac{3}{\epsilon} + 1} < \frac{N + N'}{\frac{3}{\epsilon}} \leq \frac{2\epsilon N}{3}. \tag{3.6}$$

The negative error introduced by the decrease operations in Case 3 of Step 2 is at most $\frac{2\epsilon N}{3}$.

[2] N' can be any positive integer less than or equal to N.
[3] All estimates in the $\frac{3}{\epsilon}$ entries are decreased by 1, and the new item is not counted.

Consequently, for any item i, we get

$$0 \leq f_i - \hat{f}_i \leq \frac{\epsilon N}{3} - 1 + \frac{2\epsilon N}{3} < \epsilon N. \tag{3.7}$$

Therefore, SNAPSHOT-BASIC can maintain ϵ-approximate frequency estimation over sliding windows with size at most N. Also, this algorithm only makes one-sided error.

Space Requirement:

SNAPSHOT-BASIC keeps an item list L which has at most $\frac{3}{\epsilon}$ entries. Also, some snapshots are stored. For each item in the list, there may be two kinds of snapshots. One kind is the current snapshot. This kind of snapshots for all items is no more than the number of item entries which is $\frac{3}{\epsilon}$. Another kind is the old snapshots which represent position information of at least $\frac{\epsilon N}{3}$ identical items. Step 1 guarantees that no expired snapshot exists in the space, so this kind of snapshots is no more than $\frac{N}{\epsilon N/3} = \frac{3}{\epsilon}$. Therefore, there are at most $\frac{3}{\epsilon}$ item entries and $\frac{6}{\epsilon}$ snapshots, and the total space requirement of SNAPSHOT-BASIC is $O(\frac{1}{\epsilon})$. □

The proof also shows that for any window with size less than N, if it is included in the current N-sized window, SNAPSHOT-BASIC can return the frequency estimates with an error less than ϵN.

We can also record the number of decrease operations (in Case 3 of Step 2), and use this information to refine the frequency estimates. Suppose an item i's oldest valid snapshot is $s_{i,1}$, we can count the number of decrease operations after $s_{i,1}$ and add this number to $\hat{f}_i$ when querying its frequency. However, this additional information cannot improve the error bound, therefore we do not utilize such information about performing decrease operations in our algorithms.

Now we explain why we bound the number of item entries to $\frac{3}{\epsilon}$, and snapshoot positions for each $\frac{\epsilon N}{3}$ identical items.[4] Suppose we bound the item entries to $\frac{x}{\epsilon}$, and snapshoot positions for each $\frac{\epsilon N}{y}$ identical items. SNAPSHOT-BASIC totally needs $\frac{x}{\epsilon}$ item entries and $\frac{x+y}{\epsilon}$ snapshot entries in the worst case, therefore the total number of entries is $\frac{2x+y}{\epsilon}$.

[4] If $\frac{3}{\epsilon}$ is not an integer, we bound the number of item entries to $\lceil \frac{3}{\epsilon} \rceil$; If $\frac{\epsilon N}{3}$ is not an integer, we snapshoot positions for each $\lfloor \frac{\epsilon N}{3} \rfloor$ identical items.

Theorem 5. *To guarantee the ϵ-approximation error in* SNAPSHOT-BASIC*, the total number of entries $\frac{2x+y}{\epsilon}$ reaches the minimum value when $x = y = 3$.*

Proof. We want to minimize the objective function

$$f(x, y) = 2x + y. \tag{3.8}$$

We have positive constraints for x and y that $x, y > 0$. Similar to the proof of Theorem 4, the estimation error must be bounded by

$$\frac{\epsilon N}{y} + \frac{2N}{\frac{x}{\epsilon}} \leq \epsilon N, \tag{3.9}$$

and we get

$$\frac{1}{y} + \frac{2}{x} \leq 1. \tag{3.10}$$

Therefore

$$y \geq \frac{x}{x-2}, \tag{3.11}$$

and

$$x > 2. \tag{3.12}$$

When $y = \frac{x}{x-2}$, $f(x, y)$ can reach its minimum value. So

$$f(x, y) \geq 2x + \frac{x}{x-2} = \frac{x(2x-3)}{x-2} = g(x). \tag{3.13}$$

$$\frac{dg(x)}{dx} = \frac{2x^2 - 8x + 6}{(x-2)^2} = \frac{2(x-1)(x-3)}{(x-2)^2}. \tag{3.14}$$

When $x = y = 3$, $\frac{dg(x)}{dx} = 0$, and $f(x, y)$ reaches its feasible minimum value 9. □

Therefore, when we bound the number of item entries to $\frac{3}{\epsilon}$, and snapshoot positions for each $\frac{\epsilon N}{3}$ identical items, we use the minimum number of entries in the worst case. For instance, if we set $x = 4$, then $y = 2$, and $x + 2y = 10$, which is larger than the minimum value. In practice, the item entry and snapshot entry may spend different sizes of space. For instance,

if we want to estimate the number of packets from each source IP in an IPv6 network over a sliding window with 20-bit size, then each item identifier will use 128 bits, while a snapshot entry will use about 20 bits. In this case, we can rewrite the objective function by multiplying the space size of different entries, and calculate the optimal parameters of x and y. We also need consider the data structure overhead for each entry in the objective function.

Although Snapshot-Basic can maintain ϵ-approximate frequency estimation over sliding windows using $O(\frac{1}{\epsilon})$ entries, it needs $O(\frac{1}{\epsilon})$ operations for each arrival item in the worst case. Both Step 1 and Case 3 of Step 2 need $O(\frac{1}{\epsilon})$ operations. Even though we can use a binary tree to manage the item list, the decrease operation in Case 3 of Step 2 still needs $O(\frac{1}{\epsilon})$ operations in the worst case. Therefore, Snapshot-Basic is not adequate to process large data streams with high rates.

3.2.2.2 Advanced Algorithm

We propose an advanced algorithm Snapshot-Advanced which can maintain ϵ-approximate frequency estimation over sliding windows using $O(\frac{1}{\epsilon})$ entries and $O(1)$ operations. It uses more sophisticated data structure to maintain item entries and snapshot entries such that all operations can run in $O(1)$ time.

First, we use a hash table T to manage all item entries. Then the operation of finding the existence of an item can be performed in $O(1)$ time.

To reduce the running time of inserting a new snapshot, deleting an expired snapshot, or finding a given snapshot, all snapshot entries of all items are inserted into a doubly-circularly-linked list S. When the current position needs to be snapshot, it is inserted to the head of S. When the oldest snapshot entry expires, it is exactly the tail of S. Therefore, S is an automatically sorted list, with the oldest snapshot in the tail, and the newest snapshot in the head. All operation on the list S, i.e., inserting the current snapshot, deleting the oldest snapshot, and deleting a given snapshot, can be performed in $O(1)$ time.

The most difficult operation in terms of time complexity is how to decrease each frequency estimate by 1 and delete the items and snapshots that have no need to keep (which corre-

sponding counts go down to 0). To solve this issue, we first designate snapshots into two groups, ***complete snapshots*** and ***partial snapshots***. A snapshot is *complete* if the item's counter increases at least $\frac{\epsilon N}{3}$ after (including) that snapshot. A snapshot is *partial* if the item's counter dose not increase as many as $\frac{\epsilon N}{3}$ after that snapshot. In the previous data stream example shown in Figure 3.5, the up arrows $\uparrow_c$ and $\uparrow_p$ indicate a complete snapshot and a partial snapshot respectively (when looking from position 22). Notice that all complete snapshots were partial snapshots at the beginning. But once a partial snapshot updates to a complete snapshot, it will remain complete until it is expired and deleted no matter whether the decrease operation happens or not later.

For each item i in hash table T, we use three auxiliary counters, $cntc_i$, $cnto_i$ and $cntb$, combined together to represent its frequency estimate. Here $cntc_i$ is the number of complete snapshots that item i has, and $cnto_i$ is an offset counter which records the offset to $cntb$, where $cntb$ is a shared base counter for all items which is first set to 0 before processing the data stream. Once a decrease operation happens, instead of decreasing all items' offset counters by 1, $cntb$ is increased by 1 mod $\frac{\epsilon N}{3}$. If an arrival item i is an old item without a partial snapshot or a new item, its offset counter is set by

$$cnto_i = (cntb + 1) \bmod \frac{\epsilon N}{3}. \tag{3.15}$$

If an arrival item i is an old item with a partial snapshot, its offset counter is increased by 1 mod $\frac{\epsilon N}{3}$. The frequency estimate of an item i is calculated by

$$\hat{f}_i = cntc_i \cdot \frac{\epsilon N}{3} + ((cnto_i - cntb) \bmod \frac{\epsilon N}{3}). \tag{3.16}$$

All partial snapshots which have the same offset counter value $cnto_x$ are grouped into a doubly-linked list P_{cnto_x}, which is called a *local* partial snapshot list. To save space, we keep only one $cnto_x$ for each P_{cnto_x} instead of keeping $cnto_x$ for each item in this list. All such local lists are managed by an automatically sorted doubly-circularly-linked list P. That is, all the heads in all local partial snapshot lists are linked in P, which is called a *global* partial

snapshot list. If a partial snapshot is in a local list P_{cnto_x}, we say that it is also in global list P. As mentioned above, both complete snapshots and partial snapshots are inserted into the snapshot list S. However, only partial snapshots are inserted into P.

At the beginning, both item table T and snapshot list S are empty. $cntb$ is set to 0, and the global partial snapshot list P has an entry P_{cntb} which is empty. The head of P always points to P_{cntb} no matter whether P_{cntb} is empty or not, and P_{cntb} is called *garbage* list. Instead of bounding the number of item entries in SNAPSHOT-BASIC, we bound the number of partial snapshot entries in P to $\frac{3}{\epsilon}$ in SNAPSHOT-ADVANCED. Let $|P_{cnto_x}|$ denote the number of partial snapshot entries in P_{cnto_x} which have offset counter value $cnto_x$, then $\sum_{cnto_x=0}^{\frac{\epsilon N}{3}-1} |P_{cnto_x}| \leq \frac{3}{\epsilon}$.

Step 1: Delete expired snapshot and item.

Let s denote the snapshot on the tail of S, and suppose s belongs to item j.
If s is not expired, goto Step 2.
Delete s from S.
If s is a complete snapshot, decrease $cntc_j$ by 1.
If s is a partial snapshot, delete it from P_{cnto_j}.
If $cntc_j = 0$ and j has no partial snapshot, delete item j from T.

Step 2: Process the arrival item i.

Case 1: i is an old item with partial snapshot.

Delete its partial snapshot entry from P_{cnto_i}.
Let $k = (cnto_i + 1) \bmod \frac{\epsilon N}{3}$.
If $k \neq cntb$, insert this entry into P_k;
Otherwise, increase $cntc_i$ by 1 and set this entry as complete snapshot.

Case 2: i is an old item without partial snapshot or a new item, and P is not full.

If i is a new item, create an item entry for item i with $cntc_i = 0$, and insert it into T.
Create a partial snapshot entry recording the current position, and insert it into the head of S and $P_{(cntb+1) \bmod \frac{\epsilon N}{3}}$.

Case 3: i is an old item without partial snapshot or a new item, and P is full.

If $P_{(cntb+1) \bmod \frac{\epsilon N}{3}}$ exists, move head of P to it and delete P_{cntb}.
Increase $cntb$ by 1 mod $\frac{\epsilon N}{3}$.

Step 3: Garbage Collection.

If P_{cntb} is not empty, delete its head entry from P_{cntb} and S.
If the corresponding item has no complete snapshot, delete this item entry from T.

Figure 3.9 SNAPSHOT-ADVANCED Algorithm Description

Figure 3.9 gives the algorithm description of SNAPSHOT-ADVANCED. Figure 3.7 and Figure 3.8 show an example of how SNAPSHOT-ADVANCED works. In this example, $N = 18$ and $\epsilon = \frac{1}{2}$, and the input data stream is shown in Figure 3.5. The partial snapshot list will keep at most 6 entries, and positions are snapshot for each 3 identical items. The up arrows "$\uparrow_c$" and '$\uparrow_p$" in Figure 3.5 indicate that the corresponding snapshot is a complete snapshot or a partial snapshot respectively (when looking from position 22). Figure 3.7(a) shows the hash table T when $n = 19$, and there are 6 item entries in the hash table T, 5 partial snapshot entries in the global partial snapshot list P, and totally 8 snapshot entries in the snapshot list S. We do not draw the sorted snapshot list S to save paper space. These 5 partial snapshots are grouped according to their offset counters as shown in 3.8(a). Now $cntb = 0$, the garbage list P_0 is empty and its next list is P_1. When the 20th item i_7 arrives, no snapshot expires. The new item i_7 is inserted into the hash table T as shown in Figure 3.7(b). As its hash value is equal to that of i_3, i_7 and i_3 are in the same item list. A new partial snapshot entry ps_{i_7} is inserted into the head of the corresponding local partial snapshot list P_1 as shown in Figure 3.8(b). It is also inserted into the head of snapshot list S. Now there is no room left in the partial snapshot list. When the 21st item i_8 arrives, the complete snapshot $s_{i_2,1}$ on position 3 which belongs to i_2 expires and is deleted from the tail of snapshot list S, and $cntc_{i_2}$ is decreased by 1. Because i_8 is not present in the item list and there is no room for creating a new partial snapshot entry, $cntb$ is increased by 1 mod $\frac{\epsilon N}{3}$. The down arrow "$\Downarrow$" in Figure 3.5 shows that there is such a "decrease operation" in this position. The garbage list updates to P_1, and P_0 is deleted. The partial snapshot ps_{i_7} in the head of P_1 is deleted. ps_{i_7} is also deleted from snapshot list S. The other 3 partial snapshots in the garbage list will be released one by one when the following 3 items arrive. i_7 is deleted from the item list as $cntc_{i_7} = 0$ and it has no partial snapshot. The items and partial snapshots surrounded by dotted lines in Figure 3.7(c) and Figure 3.8(c) show that these entries should be released.

Notice that, when "decrease operation" happens, not all present items in T will decrease their estimates by 1. Only the items with a partial snapshot perform this operation. For example, in Figure 3.5, there is a snapshot on position 7 for item i_5. It converts from a partial

snapshot to a complete snapshot after the 17th item arrives, and $\hat{f}_{i_5} = 3$. When the 21st item arrives, i_5 has no partial snapshot and does not decrease its estimate.

Now we prove the correctness of the SNAPSHOT-ADVANCED algorithm, and give the space requirement and running time.

Theorem 6. *For any item i, no matter whether it appears in the data stream or not,* SNAPSHOT-ADVANCED *can maintain ϵ-approximate frequency estimation over sliding windows with size at most N. Furthermore,* SNAPSHOT-ADVANCED *uses $O(\frac{1}{\epsilon})$ space and $O(1)$ running time when processing each arrival item and answering any query.*

Proof. Approximation:

The proof of approximation correctness is similar to that in Theorem 4 for SNAPSHOT-BASIC. The error of frequency estimate still comes from two sources: One is the operation in Step 1, and the other is the operation in Case 3 of Step 2. Similarly, the error introduced by Step 1 is a negative error at most $\frac{\epsilon N}{3} - 1$. When Case 3 of Step 2 happens, all items with partial snapshots must decrease their frequencies by 1, and the arrival item cannot be counted. Therefore, this operation will introduce a negative error equal to 1 for certain items. With the same deduction, such an operation can only perform at most $\frac{2\epsilon N}{3}$ times in any window with size N or less. Finally, for any item i, we get

$$0 \leq f_i - \hat{f}_i < \epsilon N \tag{3.17}$$

Consistence:

To keep the constant time complexity in the garbage collection step, each time we only release one garbage entry. Therefore we must consider the consistence issue. First, *cntb* will not change before all garbage entries are released. Each time Step 3 will release a partial snapshot entry which can be used to insert the next new item, so Case 3 of Step 2 will never happen before the garbage list is empty. It is possible that an item's partial snapshot is in the garbage list and waiting for release while an identical item arrives. In this scenario, this partial snapshot is removed from the garbage list to the list with offset counter value $(cntb + 1) \bmod \frac{\epsilon N}{3}$. A

partial snapshot can be removed out from the garbage list, however, no partial snapshot can enter the garbage list. The reason is that once a partial snapshot's offset counter updates from $(cntb - 1) \bmod \frac{\epsilon N}{3}$ to $cntb$, according to Case 1 of Step 2, this partial snapshot converts to a complete snapshot and deleted from the partial snapshot list.

Space Requirement:

SNAPSHOT-ADVANCED keeps snapshot entries in linked lists and item entries in a hash table. The number of partial snapshots is bounded to $\frac{3}{\epsilon}$ (including the partial snapshots in the garbage list). The number of complete snapshots is no more than $\frac{N}{\epsilon N/3} = \frac{3}{\epsilon}$ as Step 1 guarantees that no expired snapshot exists in the space. Also, each item entry has at least one snapshot, so the number of item entries is no more than the number of all snapshots. Therefore, the total space requirement of SNAPSHOT-ADVANCED is $O(\frac{1}{\epsilon})$.

Complexity:

As discussed above, all operations listed in Figure 3.9 can be performed in $O(1)$ time and there is no loop in the algorithm, therefore the complexity of SNAPSHOT-ADVANCED to process each arrival item is just $O(1)$.

To answer an arbitrary query concerned about item i, if item i cannot be found in the hash table, then we return 0 as its frequency estimate. If the queried item i is present in the hash table, we can easily retrieve $cntc_i$, $cnto_i$ and $cntb$ in $O(1)$ time, and use equation (3.16) to calculate its frequency estimate. Therefore, the complexity of SNAPSHOT-ADVANCED to answer a query is $O(1)$. □

3.2.3 Experimental Evaluation

In our experimental studies, we use real world Internet traffics provided by CAIDA [6] to evaluate the performance of SNAPSHOT-BASIC and SNAPSHOT-ADVANCED. The data set is from one of CAIDA's OC48 traces[5], which records all packets' header information collected at one large Internet Service Provider (ISP) in San Jose, California on April 24, 2003. The OC48 network trace we used in our experiments have totally 84,579,312 packets from 225,488 unique

[5]OC48 is a network line with transmission speeds of up to 2488.32 Mbit/s.

source IP addresses. The packet number distribution is shown in Figure 3.10, which indicates that most source IPs have small number of packets and some elephant IPs hide in them. For instance, more than half IPs have less than 10 packets.

All experiments were run on a computer with 3.2GHz Pentium IV CPU and 2GB RAM, which operation system is Windows XP Professional. All algorithms are implemented using C language.

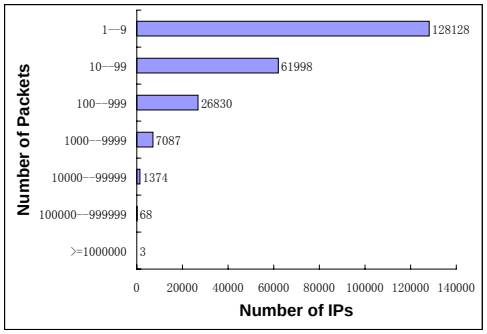


Figure 3.10 Packet Number Distribution.

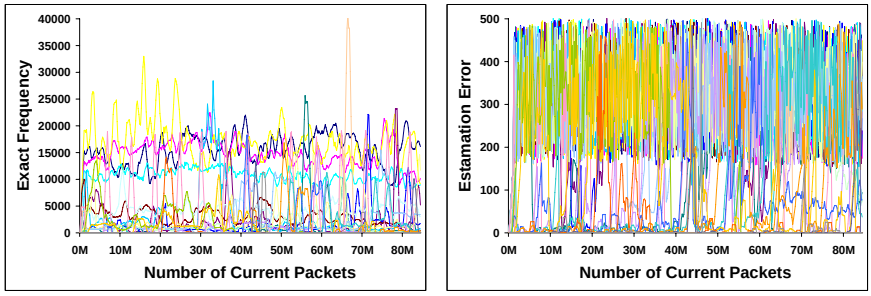
3.2.3.1 Estimation Error

To evaluate the correctness of our algorithms, we set the window size N to 1,000,000, and the relative estimation error ϵ to 0.001. Therefore, we expect that the estimation error of our algorithms is within 1000 ($= \epsilon N$) during last 1,000,000 packets at any position for any source IP. Figure 3.11 shows the experimental results. Since it is hard and unnecessary to draw all 225,488 source IPs' frequencies, to make readable figures, we only show the 24 frequent source IPs which have at least 10,000 packets in a certain window of 1,000,000 packets when the sliding window slides over the 84,579,312 packets. [6] Figure 3.11(a) shows the exact frequencies of the 24 frequent source IPs in sliding windows. The X-axis denotes the last packet's index number in current sliding window, and here 1M equals 1,000,000; The Y-axis denotes the exact frequencies (number of packets from a distinct frequent source IP). As shown in the figure, the most frequent source IP has about 41,000 packets in a sliding window.

The estimation errors of these 24 frequent IP flows using SNAPSHOT-BASIC and SNAPSHOT-

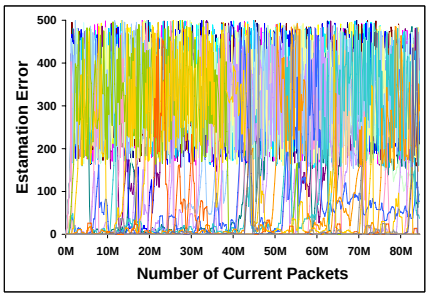
[6] Other source IP flows have similar pattern as these 24 frequent IP flows.

ADVANCED are shown in Figure 3.11(b) and 3.11(c) respectively. The X-axes of them denote the same as the X-axis in Figure 3.11(a); The Y-axes denote the estimation error $(f_i - \hat{f}_i)$, where f_i is the exact frequency of source IP i and $\hat{f}_i$ is its frequency estimation. As expected, the estimation error $(f_i - \hat{f}_i)$ is one-sided error, and the largest estimation error is about 500 which is far away from the theoretical bound ($\epsilon N = 1000$). The estimation errors of SNAPSHOT-BASIC and SNAPSHOT-ADVANCED have similar patterns. Most of the estimation errors vibrate between range [170,500]. The reason is that the "decrease operation" happens about 170 times during the range of 1,000,000 packets in this data set, and the deletion of an expired position always introduces a sudden error of $\frac{\epsilon N}{3} \approx 330$ to that item.



(a) Exact frequencies of 24 frequent items in sliding windows

(b) Estimation error of SNAPSHOT-BASIC



(c) Estimation error of SNAPSHOT-ADVANCED

Figure 3.11 Experimental Results ($N = 1,000,000$. $\epsilon = 0.001$)

3.2.3.2 Space Requirement

Although there are many linked lists in SNAPSHOT-BASIC and SNAPSHOT-ADVANCED, and an additional hash table in SNAPSHOT-ADVANCED, there are only 2 kinds of entries maintained in memory: item entries and snapshot entries. We set $N = 1,000,000$ and calculated the

maximum number of each kind of entries in memory with different ϵ. The results shown in Table 3.1 support that both SNAPSHOT-BASIC and SNAPSHOT-ADVANCED require $O(\frac{1}{\epsilon})$ space.

Table 3.1 Space Requirement

$\frac{1}{\epsilon}$	SNAPSHOT-BASIC		SNAPSHOT-ADVANCED	
	max # of items	max # of snapshots	max # of items	max # of snapshots
10^2	300	323	303	322
$10^{2.25}$	534	591	538	591
$10^{2.5}$	949	1098	955	1102
$10^{2.75}$	1688	2089	1699	2093
10^3	3000	4071	3017	4092

3.2.3.3 Running Time

We implemented the LEE-TING algorithm [65] and compared with SNAPSHOT-BASIC and SNAPSHOT-ADVANCED. In SNAPSHOT-ADVANCED, we set the size of the hash table T to $\frac{3}{\epsilon}$ (i.e., the average length of T is slightly larger than 1). We use (int32(IP address) mod sizeof(T)) as the hash function. For instance, suppose $\epsilon = 0.001$ and the size of T is 3000, an item with IP address 1.2.3.4 is hashed to the entry with index $(1 \cdot 2^{24} + 2 \cdot 2^{16} + 3 \cdot 2^8 + 4)\%3000 = 1060$. In our experiments, we set $N = 1,000,000$, and profiled the running time of these algorithms with different ϵ.

Figure 3.12 shows the results. We made the observation that both LEE-TING and SNAPSHOT-BASIC have nearly linear running time with respect to $\frac{1}{\epsilon}$, and SNAPSHOT-BASIC is faster than LEE-TING. However, the running time of them is unacceptable if we need more precise estimation with smaller ϵ. In this case, SNAPSHOT-ADVANCED has significant advantage over LEE-TING and SNAPSHOT-BASIC since its running time is nearly constant as shown in Figure 3.12.

3.2.4 Extensions

As an application, we extend our algorithms to another problem – how to estimate flow size. Here a flow is defined as a substream which items have the same item identifier. In this problem, each arrival item is associated with a positive integer (e.g., the number of bytes in

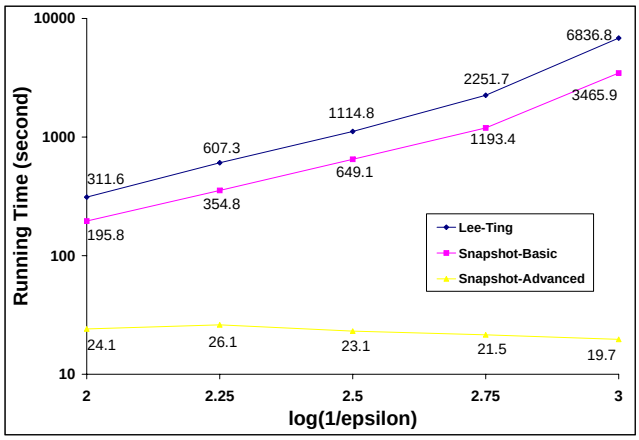


Figure 3.12 Running Time

an Internet packet). we need to estimate the accumulation of these positive numbers of each distinct item. For example, in an Internet packet stream collected from an ISP with millions of different source IP addresses, all packets from an identical source IP are combined into a flow, and each packet may contain more than one byte of payload. We are interested in how to estimate the payload bytes sent out from distinct source IP addresses.

There may be two definitions of the sliding windows in this case, and hence two problems on how to maintain ϵ-approximate frequency. The first sliding window is defined to cover the most recent N **payload bytes**, and the second is defined to cover the most recent N **packets**. Because of page limitation, we only give sketches of the solutions.

Problem 1. *Given a packet stream, a window size N and an error bound ϵ, how to estimate the size of any flow with error no more than ϵN bytes in the sliding window of the most recent N payload bytes?*

Sketch Solution: Under this sliding window definition, we can replace each packet as a series of payload bytes with adjacent position indices. For instance, we receive a packet of flow i with a payload of x bytes. Suppose the position index of the last payload byte of the previous packet is n, then these new bytes will be indexed as $n+1$, $n+2$, ..., $n+x$. The naive solution is to seem these x bytes as x identical items and call our algorithms x times. However, it is time consuming. We can just run our algorithms one time with the following changes. When checking expired snapshots in step 1, more than one snapshot may be expired and deleted.

When processing an old item, there may be more than one position that need to be snapshot (when $x > \frac{\epsilon N}{3}$). When performing decrease operation, we perform decrease operation x times, or until an entry is deleted after k times. We then insert the new item with a counter $x - k$.

Problem 2. *Given a packet stream, a window size N and an error bound ϵ, how to estimate the size of any flow with error no more than ϵN bytes in the sliding window of the most recent N packets?*

Sketch Solution: Under this sliding window definition, let c denote the maximum number of packet payload bytes. We must assume that $c << \epsilon N$. Otherwise, any deterministic algorithm must keep each packet's information in memory to maintain ϵ-approximation in the worst case. When $c << \epsilon N$, we just bound the number of flow entries to $\frac{3c}{\epsilon}$, and snapshoot positions for each $\frac{\epsilon N}{3}$ payload bytes of identical packets.

3.2.5 Conclusions

In this research, we address the problem of estimating ϵ-approximate frequency in data streams over sliding windows. Two novel deterministic algorithms, Snapshot-Basic and Snapshot-Advanced are proposed which only need $O(\frac{1}{\epsilon})$ space. Furthermore, Snapshot-Advanced is the first efficient algorithm which can achieve $O(\frac{1}{\epsilon})$ space requirement and only need $O(1)$ running time to process each item in the data stream and to answer a query. Our experimental studies show the advantages of our algorithms when processing large-scale data sets. In addition, as an application of our algorithms, we extend them to solve the problem of estimating flow size. In the future, we will continue to study the problems of gathering characteristics of data streams over different types of sliding windows.

3.3 Geometric Estimation over Sliding Windows

3.3.1 Introduction

Geometric computation has been widely studied by many researchers in different domains and utilized in many different applications in recent decades. Many geometric problems have

had optimal solutions. However, the same geometric problem may lack efficient algorithm if it is reconsidered under data stream model. Recently, data streams have received considerable attention [56, 18, 82]. If we have sufficient large space and do not have time constraints, we can precisely answer any geometric queries. However, the issue in computing geometry in large data streams is that in many cases we probably only have one chance to process each point in the data streams. We cannot store all data because of constraints in memory space or privacy issue. Therefore, in these cases we have to gather the interested geometry information with only one pass.

An even greater challenge is to compute geometry over sliding windows. An algorithm which works over sliding windows can not only gather the data streams' geometry information, but also update the geometry information by inserting new points and deleting expired points. Unfortunately, many previous geometry computation algorithms cannot work over sliding windows. In this research, we are interested in estimating diameter, convex hull and skyline over sliding windows.

3.3.1.1 Motivation

Geometric computation has many applications, such as computer graphics, computer-aided design and manufacturing (CAD/CAM), geographic information systems (GIS), integrated circuit geometry design and verification, etc. Also, geometric computation becomes an important issue in network security after distributed networks (e.g., sensor networks) are widely researched and deployed. For instance, in the early age of worm propagation, after receiving thousands of alarms from distributed network monitors, a geometric map is necessary to show which regions have been affected so that countermeasures can be executed to interrupt the worm propagation. **Diameter** can scale how far the worm has propagated, and **convex hull** can reflect the boundary of the affected hosts. Furthermore, geometric computation is required not only in the domain of geometric coordinates, but also in many other fields in network security. For instance, network logs contain a lot of hosts which volume, connection numbers, etc. are recorded, and we want to analyze and detect the dominant hosts in some

terms which should have more chance to be attackers. The **skyline** calculation can be applied to this purpose.

In recent years, some geometric computation problems are considered under sliding window models. The advantage of an algorithm which works over sliding windows is that it can get rid of the stale points and only consider the fresh points, which is meaningful in many cases. For instance, in an intrusion detection system (IDS), the current status of the network is usually more important than that of one day ago. The geometry information gathered over sliding windows may provide a more fresh and smooth view of the data stream. Recently, several geometry algorithms over sliding windows are proposed [45, 28, 68, 103]. However, the algorithms for diameter and convex hull estimation over sliding windows still need some improvement. To the best of our knowledge, skyline estimation over sliding windows still lacks efficient algorithms. In this research, we are trying to design efficient algorithms for diameter, convex hull and skyline estimation over sliding windows.

3.3.1.2 Problem Definition

ϵ-Approximate Diameter Estimation

Suppose we have a stream of points in set P. The ***diameter*** of P is defined as

$$\max_{\forall p,q \in P}(\|p-q\|), \tag{3.18}$$

i.e., the maximum Euclidean distance between any pair of points in P. Let D and $\hat{D}$ denote the true diameter and estimated diameter respectively. A diameter estimation algorithm is an **ϵ-approximate** algorithm if it guarantees

$$|D - \hat{D}| \leq \epsilon D. \tag{3.19}$$

ϵ-Approximate Convex Hull Estimation

Suppose we have a stream of points in set P. The ***convex hull*** of P is the smallest polygon that contains all points in P. Let H denote the set of all points within the true convex hull, and $\hat{H}$ denote the set of all points within the estimated convex hull. A convex hull estimation algorithm is an **ϵ-approximate** algorithm if it guarantees

$$\|H - \hat{H}\| = \max_{\forall p \in H}(\min_{\forall q \in \hat{H}}(\|p - q\|)) \leq \epsilon D, \tag{3.20}$$

where D is the diameter of set P.

ϵ-Approximate Skyline Estimation

Suppose in d-dimension, we have a stream of points in set P. For two points $a = (a_{(1)}, a_{(2)}, \ldots, a_{(d)})$ and $b = (b_{(1)}, b_{(1)}, \ldots, b_{(d)})$ in P, a ***dominates*** b if $a_{(i)} \leq b_{(i)}$ for $1 \leq i \leq d$. The ***skyline*** is the set of points which are not dominated by any other point in P. Let S denote the set of points on the true skyline, and $\hat{S}$ denote the set of points on the estimated skyline. A skyline estimation algorithm is an **ϵ-approximate** algorithm if it guarantees that

$$\|S - \hat{S}\| = \max_{\forall p \in S}(\min_{\forall q \in \hat{S}}(\|p - q\|)) \leq \epsilon D, \tag{3.21}$$

where D is the diameter of set P.

Sliding Window

A ***sliding window***, first introduced by Datar et al. [35], only contains the last N items in the data stream, which is updated once a new element comes and an old element expires. Here N is the width of the sliding window.

Problem Statement

In this research, we consider the problems stated as follows:

Given an arbitrary window size N and an error bound ϵ,

- How to maintain ϵ-approximate diameter estimation of a data stream of points over sliding windows with size N in one pass?
- How to maintain ϵ-approximate convex hull estimation of a data stream of points over sliding windows with size N in one pass?
- How to maintain ϵ-approximate skyline estimation of a data stream of points over sliding windows with size N in one pass?

3.3.1.3 Our Contributions

To our knowledge, the best existing algorithm for the problem of estimating ϵ-approximate diameter in data streams over sliding windows requires $O((\frac{1}{\epsilon})^{\frac{d+1}{2}} \log \frac{R}{\epsilon})$ space [28], where R is the ratio between the largest distance and the smallest distance of a pair of points, and d is the dimension. We first present an improved algorithm which only requires $O((\frac{1}{\epsilon})^{\frac{d+1}{2}} \log R)$ space. We then extend our algorithm to solve convex hull estimation problem over sliding windows, and prove that the exact diameter algorithm can get the ϵ-approximate convex hull estimation directly. Finally, we propose a novel algorithm to estimate skyline which requires $O(\frac{1}{\epsilon^d} \log R)$ space.

3.3.2 Diameter Algorithm

3.3.2.1 Chan and Sadjad's Previous Algorithm

We first briefly review Chan and Sadjad's algorithm on diameter estimation over sliding windows [28]. In one-dimension, they proposed an optimal algorithm to maintain the approximate maximum and minimum. As an instance, to maintain the maximum, let $Q =< q_1, q_2, \ldots, q_k >$ be a subsequence of P such that $q_1 < q_2 < \ldots < q_k$, where P is the set of input points. Let $pred_Q(p)$ denote the maximum value in Q that is at most p, and $succ_Q(p)$ denote the minimum value in Q that is at least p. Q is called a *summary sequence* of P if

1. Q is in decreasing order of arrival time.
2. For all p, $pred_Q(p)$ is not older than p if existing.

3. For all p, either $\|p - pred_Q(p)\| \leq \epsilon D_p$ or $succ_Q(p)$ is not older than p.

Here D_p denotes the diameter of all points in P which are not older than p. The summary sequence Q is enough to maintain the ϵ-approximate maximum of P. When inserting a new point p, all points in Q that are not greater than p are removed, and p is put at the beginning of Q. After $\frac{1}{\epsilon}\log R$ new points are inserted, a refine process is executed to reduce the points in Q.

Refine in Chan and Sadjad's Algorithm:

Let q_1 and q_2 be the 1st and 2nd points in Q respectively. Let $q := q_2$.
while q is not the last element of Q **do**
  Let x and y be the elements before and after q in Q.
  if $\|y - x\| \leq \epsilon\|x - q_1\|$
    then remove q from Q.
  Continue with q equal to y.

To estimate one-dimensional diameter, two similar data structures that approximate the maximum and minimum of the points are maintained. Chan and Sadjad's algorithm in one-dimension needs $O(\frac{1}{\epsilon}\log R)$ space and $O(1)$ running time in worst case (by running refine in a "lazy" mode). It is proved that this algorithm is optimal in one-dimension. To extend their algorithm to higher fixed dimensions, they use $\Theta((\frac{1}{\epsilon})^{\frac{d-1}{2}})$ lines in d-dimension which guarantee that for each vector x in d-dimension, the angle between x and some line is at most $\arccos(\frac{1}{1+\epsilon})$. The one-dimension summary sequence structure is maintained on each line by projecting all points to the line, and the maximum expansion on these lines are returned as the approximated diameter, which is an ϵ-approximate.

Chan and Sadjad observed a problem that naively projecting points to the lines can make the spread of the one-dimensional points arbitrarily big, since the distance of two projected points could be much smaller compared with their distance in d-dimension. To solve this problem, they always keep the location of the two newest points p_1 and p_2. Let $Q^{(l)} =< q_1, q_2, \ldots, q_k >$ be the summary sequence of projected points on line l. All q_i's that satisfies $\|q_i - q_1\| \leq \epsilon\|p_1 - p_2\|$ are removed before the refine algorithm, since q_1 can represent them. After the removal, the distance between q_1 and next point in Q is at least $\epsilon\|p_1 - p_2\|$, and the

refine algorithm guarantees the size of $Q^{(l)}$ is $O(\frac{1}{\epsilon}\log\frac{R}{\epsilon})$. Consequently, Chan and Sadjad's algorithm can maintain ϵ-approximate of diameter in d-dimension using $O((\frac{1}{\epsilon})^{\frac{d+1}{2}}\log\frac{R}{\epsilon})$ space and $O((\frac{1}{\epsilon})^{\frac{d-1}{2}})$ running time.

3.3.2.2 Improved Algorithm

Although Chan and Sadjad's algorithm has significant improvement compared with the algorithm in [45], it is still not optimal when applied in higher dimensions.

We propose an algorithm which only needs $O((\frac{1}{\epsilon})^{\frac{d+1}{2}}\log R)$ space. Similarly, we still maintain $\Theta((\frac{1}{\epsilon})^{\frac{d-1}{2}})$ lines in d-dimension which guarantee that for each vector x in d-dimension, the angle between x and some line is at most $\arccos(\frac{1}{1+\epsilon})$. Let L denote the set of these $\Theta((\frac{1}{\epsilon})^{\frac{d-1}{2}})$ lines. The one-dimension summary sequence structure is maintained on each line $l \in L$ by projecting all points to the line l, and the maximum expansion on these lines are returned as the approximated diameter, which is an ϵ-approximate. However, we use a different refine process which is more efficient.

Refine:

Let p_1 and p_2 be the 1st and 2nd points in P respectively. Let q_1 and q_2 be the 1st and 2nd projections in Q respectively. Let $q := q_2$.
while q is not the last element of Q **do**
  Let x and y be the elements before and after q in Q.
  if $\|y-x\| \le \max(\epsilon\|x-q_1\|, \epsilon\|p_1-p_2\|)$
    then remove q from Q.
  Continue with q equal to y.

Figure 3.13 shows an example of how our refine process works. All points are projected onto line l, and suppose that currently $Q^{(l)} =< q_1, q_2, \ldots, q_9 >$. During refine, projections q_2 and q_3 are removed because $\|q_4 - q_1\| \le \epsilon\|p_1 - p_2\|$. Similarly, projection q_5 is removed because $\|q_6 - q_4\| \le \epsilon\|q_4 - q_1\|$, and projection q_8 is removed because $\|q_9 - q_7\| \le \epsilon\|q_7 - q_1\|$. After refine, $Q^{(l)} =< q_1, q_4, q_6, q_7, q_9 >$.

Now we prove the correctness of our algorithm, and give the space requirement and running time.

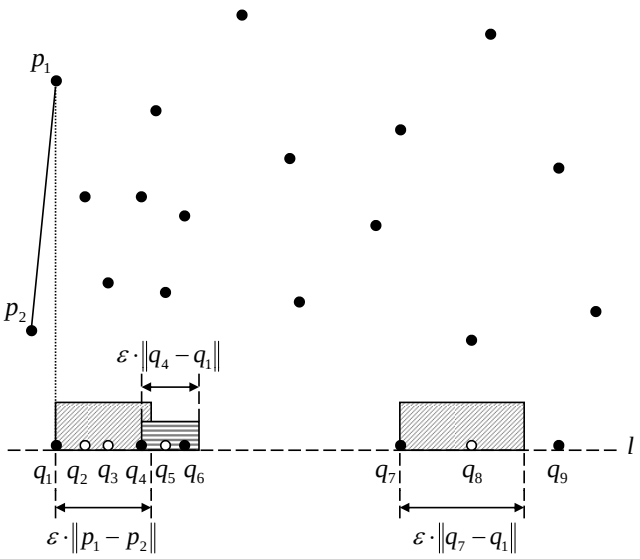


Figure 3.13 An Example of How Refine Process Works

Theorem 7. *Our algorithm can maintain ϵ-approximate diameter estimation over sliding windows in d-dimension. Furthermore, it uses $O((\frac{1}{\epsilon})^{\frac{d+1}{2}} \log R)$ space[7], and the worst running time to process a new point is $O((\frac{1}{\epsilon})^{\frac{d-1}{2}})$.*

Proof. Approximation:

We prove that our algorithm can maintain a sequence Q with the three properties of a summary sequence.

For property 1, suppose that before inserting a projection p, the current Q is in descendant order of arrival time. Since all points in Q that are not greater than p are removed, and p is put at the beginning of Q, Q remains descendant order. The following refine only removes points from Q, therefore Q is always in decreasing order of arrival time.

Property 2 is also obviously true in our algorithm, since any point's old predecessor can only be replaced by newer points.

Now we consider property 3. First, the insertion of new points cannot destroy this property. If a projection p's successor is removed by the insertion of a new projection, then the new projection will be p's new successor which is newer than p. If p's predecessor is removed by

[7] *Since the space requirement is independent with the sliding window size N, it is not necessary to delete expired points. This is also true for our convex hull algorithm and skyline algorithm.*

the insertion of a new projection, then the new projection will be p's new predecessor which is closer to p than previous one. Second, the refine cannot destroy this property. Suppose p's predecessor or successor is removed during refine. Let x and y be p's current predecessor and successor respectively. Our algorithm guarantees that $\|y - x\| \leq \max(\epsilon\|x - q_1\|, \epsilon\|p_1 - p_2\|)$. Since $\|x - q_1\| \leq D_p$, and $\|p_1 - p_2\| \leq D_p$, we have

$$\|p - x\| \leq \|y - x\| \leq \epsilon D_p. \tag{3.22}$$

Therefore, property 3 is still true.

Consequently, our algorithm guarantees that Q is a summary sequence. Therefore, suppose projection p is exactly the maximum value in all projections not older than p on line $l \in L$, we can use its predecessor to approximate p with error bounded by ϵD_p. Similarly, we can maintain another sequence which can approximate the minimum value on line $l \in L$. Suppose projections p and q are the maximum value and minimum value in all projections on line l in current sliding window respectively, then our algorithm can return p' and q' to represent p and q respectively, and

$$0 \leq \|p - q\| - \|p' - q'\| = \|p - p'\| + \|q - q'\| \leq 2\epsilon D, \tag{3.23}$$

where D is the diameter of all points within current sliding window.

Let $p_a, p_b \in P$ be the furthest pair of points in P, and $\|p_a - p_b\| = D$. Suppose line $l \in L$ has the least angle θ to line segment $\overline{pq}$. Then $\theta \leq \arccos(\frac{1}{1+\epsilon})$. Let p_{a_l} and p_{b_l} denote the projection of p_a and p_b on line l. Suppose projections p and q are the maximum value and minimum value in all projections on line $l \in L$ in current sliding window respectively. We get

$$\begin{aligned}
\|p_a - p_b\| &= \frac{\|p_{a_l} - p_{b_l}\|}{\cos\theta} \leq (1+\epsilon)\|p_{a_l} - p_{b_l}\| && (3.24)\\
&\leq (1+\epsilon)\|p - q\| && (3.25)\\
&\leq (2\epsilon + 2\epsilon^2)D + (1+\epsilon)\|p' - q'\| && (3.26)\\
&= (2\epsilon + 2\epsilon^2)\|p_a - p_b\| + (1+\epsilon)\|p' - q'\|. && (3.27)
\end{aligned}$$

And

$$\|p' - q'\| \geq \frac{1 - 2\epsilon - 2\epsilon^2}{1 + \epsilon}\|p_a - p_b\|. \tag{3.28}$$

We get

$$\begin{aligned} 0 \leq \|p_a - p_b\| - \|p' - q'\| &\leq \epsilon\frac{3 + 2\epsilon}{1 + \epsilon}\|p_a - p_b\| && (3.29)\\ &\leq 3\epsilon\|p_a - p_b\| && (3.30)\\ &= O(\epsilon)\|p_a - p_b\| && (3.31)\\ &= O(\epsilon)D. && (3.32) \end{aligned}$$

Since our algorithm can return $\|p' - q'\|$ as the approximated diameter, it is an ϵ-approximate algorithm.

Space Requirement:

To maintain maximum projection on line $l \in L$ in our algorithm, after running refine, suppose $Q^{(l)} =< q_1, q_2, \ldots, q_k >$. When $1 \leq i \leq k-2$, for any two projections q_i and q_{i+2} that have one projection between them, we have

$$\|q_{i+2} - q_i\| > \max(\epsilon\|q_i - q_1\|, \epsilon\|p_1 - p_2\|). \tag{3.33}$$

where p_1 and p_2 are the 1st and 2nd points in P respectively. Therefore, on line l, when $\|q_i - q_1\| \leq \|p_1 - p_2\|$, we have

$$\|q_{i+2} - q_i\| > \epsilon\|p_1 - p_2\|. \tag{3.34}$$

Consequently, on the line segment between q_1 and $q_1 + \|p_1 - p_2\|$, there are at most $\frac{2}{\epsilon} + 1 = O(\frac{1}{\epsilon})$ points after refine process.

Let q_j be the first projection after $q_1 + \|p_1 - p_2\|$. Then for each $j < i \leq k-2$, we have

$$\|q_{i+2} - q_i\| > \epsilon\|q_i - q_1\|. \tag{3.35}$$

Therefore, with a factor $(1+\epsilon)$, the expansion $\|q_i - q_1\|$ exponentially increases with the number of projection pairs, and the base is at least $\|p_1 - p_2\|$. Consequently, we have

$$k - j \leq 2\log_{1+\epsilon} \frac{D}{\|p_1 - p_2\|} \leq 2\log_{1+\epsilon} R = O(\frac{1}{\epsilon}\log R). \tag{3.36}$$

Since j is at most $O(\frac{1}{\epsilon})$, there are at most $O(\frac{1}{\epsilon}\log R)$ projections on each line $l \in L$. The number of lines in L are $\Theta((\frac{1}{\epsilon})^{\frac{d-1}{2}})$. Therefore, our algorithm uses $O((\frac{1}{\epsilon})^{\frac{d+1}{2}}\log R)$ space. □

Running Time:

The proof is similar to that of Theorem 1 in [28], and we skip it to save paper.

3.3.3 Convex Hull Estimation

Our diameter estimation algorithm can be directly applied to estimate convex hull in sliding windows. For convex hull problem, we maintain a similar data structure on $\Theta((\frac{1}{\epsilon})^{\frac{d-1}{2}})$ lines, and just return the points which are extremes in any line as the vertex of the estimated convex hull.

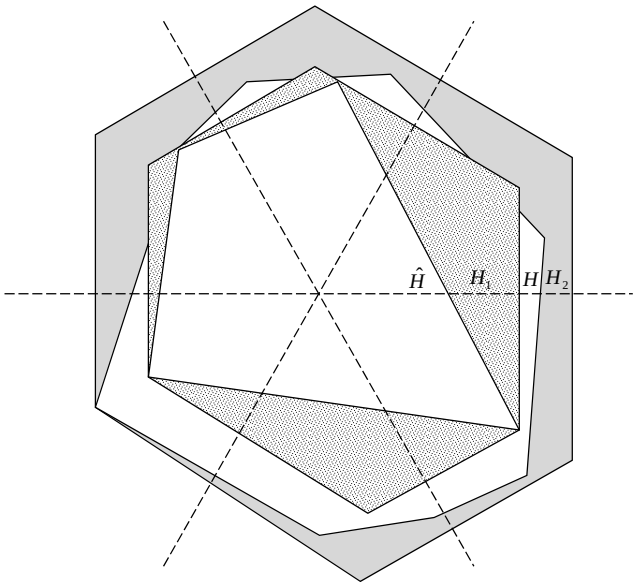


Figure 3.14 Example of H, $\hat{H}$, H_1 and H_2 in Two-Dimension

Now we prove the correctness of our algorithm, and give the space requirement and running

time.

Theorem 8. *Our algorithm can maintain ϵ-approximate convex hull estimation over sliding windows. Furthermore, it uses $O((\frac{1}{\epsilon})^{\frac{d+1}{2}} \log R)$ space, and the worst running time to process a new point is $O((\frac{1}{\epsilon})^{\frac{d-1}{2}})$.*

Proof. Since the convex hull algorithm is exactly the same as the diameter algorithm, it has the identical space requirement and running time. We only need to prove that this algorithm can maintain ϵ-approximate convex hull estimation over sliding windows.

Let H be the real convex hull in current sliding window, and $\hat{H}$ be the estimated convex hull using our algorithm. Obviously, $\hat{H} \subseteq H$. For each line $l \in L$, let p be its extreme, and we draw a supporting line (or plane) of l through p, and these supporting lines (or planes) construct a convex hull H_1 which contains $\hat{H}$. We expand H_1 by moving each edge (or plane) out with distance $\frac{\epsilon}{1-\epsilon}\hat{D}$ and let H_2 denote this new convex hall. From Theorem 7, we know that $D - \hat{D} = O(\epsilon)D$, and we assume that $D - \hat{D} \leq \epsilon D$. We have

$$\hat{D} \geq (1 - \epsilon)D, \tag{3.37}$$

and $\frac{\epsilon}{1-\epsilon}\hat{D} \geq \epsilon D$. Therefore, any vertex on H must be included in H_2, and

$$\hat{H} \subseteq H \subseteq H_2. \tag{3.38}$$

Obviously,

$$\|H - \hat{H}\| \leq \|H_2 - \hat{H}\| \leq \|H_1 - \hat{H}\| + \|H_2 - H_1\|. \tag{3.39}$$

If we can prove that the maximum distance between $\hat{H}$ and H_2 satisfies $\|H_2 - \hat{H}\| = O(\epsilon)$, then our algorithm can maintain ϵ-approximate convex hull. Figure 3.14 shows an example of H, $\hat{H}$, H_1 and H_2 in two-dimension.

We first consider $\|H_1 - \hat{H}\|$. Since Hershberger and Suri [57] have proved that the maximum gap between $\hat{H}$ and H_1 is $O(\epsilon)$ in two-dimension, we only consider higher-dimensional spaces. For any line l' in d-dimensional space, let θ be the minimum angle to a line $l \in L$. Obviously,

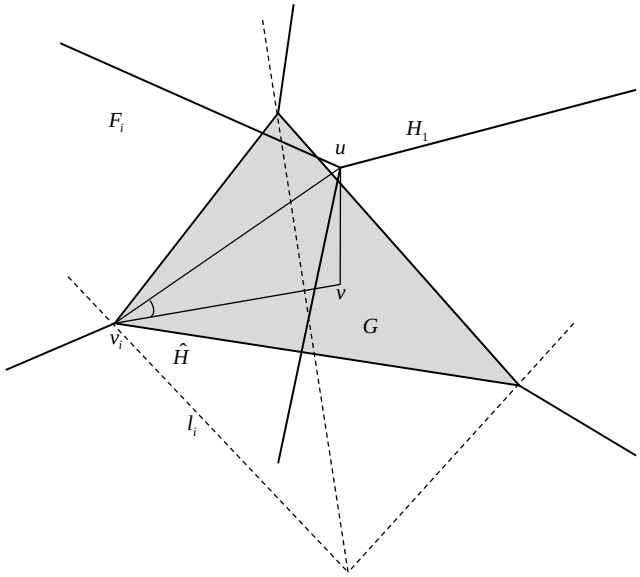


Figure 3.15 Example of $\hat{H}$ and H_1 in Three-Dimension

$\theta = O(\epsilon)$. For each plane G of $\hat{H}$, it has exact d vertices $v_1, v_2, \cdots, v_d$, which are extremes at d different directions. From each v_i, we draw a supporting plane F_i perpendicular to its extreme direction. These d supporting planes generate an intersection point u which is a vertex in H_1. Figure 3.15 shows an example in three-dimension. Obviously, the angle between G and any supporting plane is at most θ. Let v be the projection of u on plane G. Since $\overline{uv} \perp G$, and v_i is an intersection of G and the supporting plane F_i, then $\angle uv_iv$ is at most the angle between G and F_i. Therefore, $\angle uv_iv \leq \theta$. Since $\overline{vv_i} \in \hat{H}$, we have $\|v_i - v\| \leq D$. Consequently,

$$\|u - v\| \leq \|v_i - v\| \cdot \tan\theta = O(\epsilon)D. \tag{3.40}$$

Since the distance between any vertex $u \in H_1$ and $\hat{H}$ is $O(\epsilon)D$, we get $\|H_1 - \hat{H}\| = O(\epsilon)D$.

Now we consider $\|H_2 - H_1\|$. For each vertex $u_i \in H_1$, it has a corresponding vertex $w_i \in H_2$. Then

$$\|H_2 - H_1\| \leq \max(\|w_i - v_i\|). \tag{3.41}$$

The length of the projection of line segment $\overline{v_iw_i}$ on any line $l \in L$ is at most $\frac{\epsilon}{1-\epsilon}\hat{D}$ according to the construction of H_2. Since there exists a line $l \in L$ which has a angle of at most θ with

line segment $\overline{v_i w_i}$, we have

$$\|w_i - v_i\| \leq \frac{\frac{\epsilon}{1-\epsilon}\hat{D}}{\cos\theta} = O(\epsilon)D. \tag{3.42}$$

Therefore, $\|H_2 - H_1\| = O(\epsilon)D$, and $\|H_2 - \hat{H}\| = O(\epsilon)D$.

□

3.3.4 Skyline Algorithm

To estimate skyline of the stream of input points P in sliding window, we maintain a subsequence Q of points which is in descendant order of arrival time. Let D_p be the diameter of all points in P not older than p, and $\hat{D}_p$ be the ϵ-approximation of D_p using our diameter approximation algorithm.

The ***restricted zone*** Z_p of a point $p = (p_{(1)}, p_{(2)}, \cdots, p_{(d)})$ is the zone bounded by $[\lfloor \frac{p_{(i)}}{w_p} \rfloor w_p, (\lfloor \frac{p_{(i)}}{w_p} \rfloor + 1)w_p)$ in each dimension $i = 1, 2, \cdots, d$, where

$$w_p = \frac{1}{\sqrt{d}} 2^{\lfloor \log \epsilon \hat{D}_p \rfloor} \tag{3.43}$$

is the width of p's restricted zone. Figure 3.16 shows an example of restricted zones in two-dimension.

When a new point p comes, we simply insert it to the header of Q. After $\frac{1}{\epsilon^d} \log R$ points are inserted, we run the following refine process.

Refine:
Let p be the first point in Q. Let $\hat{S} = \Phi$.
while p is a point of Q **do**
Use our diameter algorithm to update $\hat{D}_p$.
If p is dominated by current skyline $\hat{S}$, remove p from Q.
If $\hat{S}$ intersects with p's restricted zone, remove p from Q. Otherwise, insert p into $\hat{S}$, and remove any points in $\hat{S}$ which are dominated by p.
Continue with p equal to next point in Q.

Figure 3.16 shows an example of how refine works in two-dimension. Suppose after inserting several new points, $Q = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ and p_i is newer than p_j when $1 \leq p_i < p_j \leq 7$.

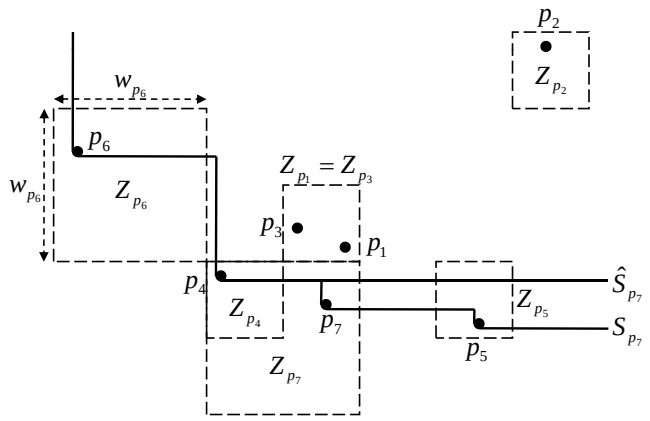


Figure 3.16 Example of Restricted Zones and How Refine Works in Two-Dimension

When the refine starts, p_1 is inserted into $\hat{S}$. Since p_2 is dominated by $\hat{S}$, p_2 is removed from Q. Also, p_3 is removed from Q because $\hat{S}$ intersects with Z_{p_3}. Then p_4 is inserted into $\hat{S}$ and p_1 is removed. Similarly, p_5 is removed from Q, and p_6 is inserted into $\hat{S}$. Notice that compared with the newer points, the width of Z_{p_6} doubles since now $\hat{D}_{p_6}$ is larger. p_7 is removed from Q because $\hat{S}$ intersects with Z_{p_7}. Finally, after refine process, $Q = \{p_1, p_4, p_6\}$, and $\hat{S}$ is the skyline decided by p_4 and p_6, although the real skyline S contains p_4 p_5, p_6 and p_7.

After refine, we have the following facts.

Fact 1. *$\hat{S}_p$ is dominated by S_p.*

Fact 2. *For any pair of points p and q in Q, suppose q is newer than p. Then either $Z_q \subseteq Z_p$ or $Z_q \cap Z_p = \Phi$.*

Proof. Let $Z_p(i) = [\lfloor \frac{p_{(i)}}{w_p} \rfloor w_p, (\lfloor \frac{p_{(i)}}{w_p} \rfloor + 1) w_p)$ be the expansion of Z_p in dimension i $(1 \leq i \leq d)$. Let $Z_q(i) = [\lfloor \frac{q_{(i)}}{w_q} \rfloor w_q, (\lfloor \frac{q_{(i)}}{w_q} \rfloor + 1) w_q)$ be the expansion of Z_q in dimension i $(1 \leq i \leq d)$. We prove that in each dimension i, either

$$Z_p(i) \cap Z_q(i) = \Phi, \tag{3.44}$$

or

$$Z_q(i) \subseteq Z_p(i). \tag{3.45}$$

Since q is newer than p, then $\hat{D}_p \geq \hat{D}_q$. Therefore, $w_p \geq w_q$. Furthermore, since $\frac{w_p}{w_q} = 2^{\lfloor \log \epsilon \hat{D}_p \rfloor - \lfloor \log \epsilon \hat{D}_q \rfloor}$, $\frac{w_p}{w_q}$ can only equal 2^x, where x is a non-negative integer. Then in dimension i, we have the following 3 cases.

Case 1: $\lfloor \frac{q_{(i)}}{w_q} \rfloor w_q < \lfloor \frac{p_{(i)}}{w_p} \rfloor w_p$

We get

$$\lfloor \frac{q_{(i)}}{w_q} \rfloor < \lfloor \frac{p_{(i)}}{w_p} \rfloor \frac{w_p}{w_q}. \tag{3.46}$$

Since both sides in inequality (3.46) are integers, we get

$$\lfloor \frac{q_{(i)}}{w_q} \rfloor + 1 \leq \lfloor \frac{p_{(i)}}{w_p} \rfloor \frac{w_p}{w_q}, \tag{3.47}$$

and

$$(\lfloor \frac{q_{(i)}}{w_q} \rfloor + 1) w_q \leq \lfloor \frac{p_{(i)}}{w_p} \rfloor w_p. \tag{3.48}$$

Therefore, $Z_p(i) \cap Z_q(i) = \Phi$.

Case 2: $\lfloor \frac{p_{(i)}}{w_p} \rfloor w_p \leq \lfloor \frac{q_{(i)}}{w_q} \rfloor w_q < (\lfloor \frac{p_{(i)}}{w_p} + 1) \rfloor w_p$

We get

$$\lfloor \frac{q_{(i)}}{w_q} \rfloor < (\lfloor \frac{p_{(i)}}{w_p} \rfloor + 1) \frac{w_p}{w_q}. \tag{3.49}$$

Since both sides in inequality (3.49) are integers, we get

$$\lfloor \frac{q_{(i)}}{w_q} \rfloor + 1 \leq (\lfloor \frac{p_{(i)}}{w_p} \rfloor + 1) \frac{w_p}{w_q}, \tag{3.50}$$

and

$$(\lfloor \frac{q_{(i)}}{w_q} \rfloor + 1) w_q \leq (\lfloor \frac{p_{(i)}}{w_p} \rfloor + 1) w_p. \tag{3.51}$$

Therefore, $Z_q(i) \subseteq Z_p(i)$.

Case 3: $\lfloor \frac{q_{(i)}}{w_q} \rfloor w_q \geq (\lfloor \frac{p_{(i)}}{w_p} + 1) \rfloor w_p$

Obviously we get $Z_p(i) \cap Z_q(i) = \Phi$.

Therefore, either $Z_q \subseteq Z_p$ or $Z_q \cap Z_p = \Phi$. □

Fact 3. *After refinement, for any two points p and q in Q, their restricted zones have no overlap, i.e., $Z_p \cap Z_q = \Phi$.*

Proof. Suppose before refinement, there are two points p and q in Q, which restricted zones have overlap. Without loss of generality, suppose q is newer than p. From Fact 2, we know that q's restricted zone is included in p's restricted zone. Our refinement will remove point p if q is still in Q. Therefore, after refinement, any two points in Q have no overlap in their restricted zones. □

Now we prove the correctness of our algorithm, and give the space requirement. We set the design of an efficient data structure for our algorithm as our future work.

Theorem 9. *Our algorithm can maintain ϵ-approximate skyline estimation over sliding windows. Furthermore, it uses $O(\frac{1}{\epsilon^d}\log R)$ space.*

Proof. Approximation:

Obviously, estimation error is introduced only in refine process when a point p is removed from Q, while p is actually on the skyline S. However, the error is bounded in our algorithm. Since $\hat{D}_p \leq D_p$, then

$$w_p = \frac{1}{\sqrt{d}} 2^{\lfloor \log \epsilon \hat{D}_p \rfloor} \leq \frac{1}{\sqrt{d}} 2^{\log \epsilon D_p} = \frac{1}{\sqrt{d}} \epsilon D_p. \tag{3.52}$$

Let $D(Z_p)$ be the diameter of Z_p, then

$$D(Z_p) = \sqrt{d} \cdot w_p \leq \epsilon D_p. \tag{3.53}$$

For any point $q \in S$, it is removed if and only if $\hat{S}_p$ intersects with Z_q. Therefore, the distance from p to $\hat{S}_p$ is at most $D(Z_p)$ which is no more than ϵD_p.

With the sliding of current window, before p expires, if $\hat{S}_p$ does not dominate p, it tends to be closer to p. Therefore, the error distance from p to $\hat{S}_p$ never exceeds ϵD_p.

Space Requirement:

Let p be the first point and D_0 be the minimum distance between any pair of points in P. Let p be the center, and we draw a set of virtual zones with width $\frac{1}{\sqrt{d}} 2^i D_0$, where

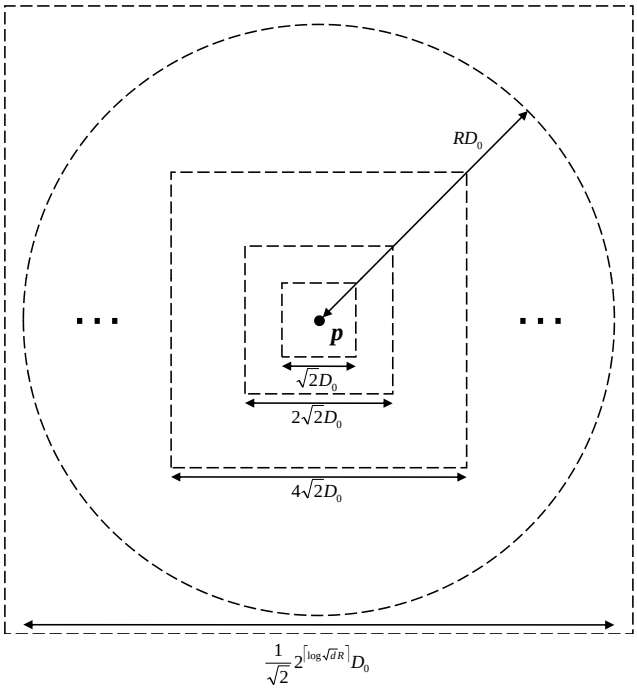


Figure 3.17 Example Virtual Zones in Two-Dimension

$i = 1, 2, \cdots, \lceil \log \sqrt{d}R \rceil$. Since R is the ratio between the largest distance and the smallest distance of a pair of points, then all points in the data stream are included in the largest zone with width $\frac{1}{\sqrt{d}}2^{\lceil \log \sqrt{d}R \rceil}D_0$. Figure 3.17 shows an example of such a set of zones in two-dimension space. The possible widths of restricted zones for all points can only be in set

$$\{\frac{1}{\sqrt{d}}2^{\lfloor \log \epsilon D_0 \rfloor}, \frac{1}{\sqrt{d}}2^{\lfloor \log \epsilon D_0 \rfloor + 1}, \cdots, \frac{1}{\sqrt{d}}2^{\lfloor \log \epsilon D_0 \rfloor + \lceil \log \sqrt{d}R \rceil - 1}\}. \tag{3.54}$$

Let $V_a(i)$ be the volume of the virtual zone with width $\sqrt{d}2^i D_0$. Let $V_b(i)$ be the volume of the restricted zone with width $\frac{1}{\sqrt{d}}2^{\lfloor \log \epsilon D_0 \rfloor + i}$. We have

$$V_a(i) = (\frac{1}{\sqrt{d}}2^i D_0)^d, \tag{3.55}$$

and

$$V_b(i) = (\frac{1}{\sqrt{d}}2^{\lfloor \log \epsilon D_0 \rfloor + i})^d. \tag{3.56}$$

Any point which restricted zone's width is $\frac{1}{\sqrt{d}}2^{\lfloor \log \epsilon D_0 \rfloor + i}$ must be within the virtual zone with

width $\frac{1}{\sqrt{d}}2^{i+1}D_0$. From Fact 3, after refinement, any two points' restricted zones cannot have overlap. Therefore, the number of points which restricted zone's width is $\frac{1}{\sqrt{d}}2^{\lfloor \log \epsilon D_0 \rfloor + i}$ is at most

$$\frac{V_a(i+1)}{V_b(i)} = (\frac{2D_0}{2^{\lfloor \log \epsilon D_0 \rfloor}})^d \leq (\frac{2D_0}{2^{\log \epsilon D_0 - 1}})^d = (\frac{4}{\epsilon})^d. \tag{3.57}$$

Therefore, when d is a small constant, the number of points after refinement is at most

$$(\frac{4}{\epsilon})^d \lceil \log \sqrt{d}R \rceil = O(\frac{1}{\epsilon^d} \log R). \tag{3.58}$$

□

Here we provide a lower bound of space requirement to maintain ϵ-approximate skyline estimation over sliding windows.

Theorem 10. *The lower bound of space requirement to maintain ϵ-approximate skyline estimation over sliding windows is $\Omega(\frac{1}{\epsilon^{d-1}} \log \epsilon R)$.*

Proof. We construct a special case to show that the lower bound of space requirement to maintain ϵ-approximate skyline estimation over sliding windows is $\Omega(\frac{1}{\epsilon^{d-1}} \log \epsilon R)$. Let p and q be the pair of points which have the minimum distance D_0, and hence the maximum distance of a pair of points is RD_0. Let p be the center, and draw a series of surface s_i with radius $\frac{1}{\epsilon}D_0$, $\frac{1}{\epsilon}(1+\epsilon)D_0 + \delta$, $\cdots$, $\frac{1}{\epsilon}(1+\epsilon)^i D_0 + i\delta$, $\cdots$, until the radius reaches $\frac{1}{\sqrt{d}RD_0}$. Here δ is a infinitely small quantity. It is guaranteed that the maximum distance of a pair of points is RD_0. On each surface s_i, we evenly distribute $O(\frac{1}{\epsilon^d})$ points. Obviously, all these points in the same surface do not dominate each other. These points are placed carefully so that the distance of any point to the skyline by other points are at least $(1+\epsilon)^i D_0 + \delta$. Suppose p is the newest point and the points on surface s_i become older with the increase of i. Therefore, all these points must be kept to guarantee ϵ-approximate. There are total $O(\log \epsilon R)$ surfaces, and each surface has $O(\frac{1}{\epsilon^{d-1}})$ points. Consequently, we have the lower bound of $\Omega(\frac{1}{\epsilon^{d-1}} \log \epsilon R)$.

□

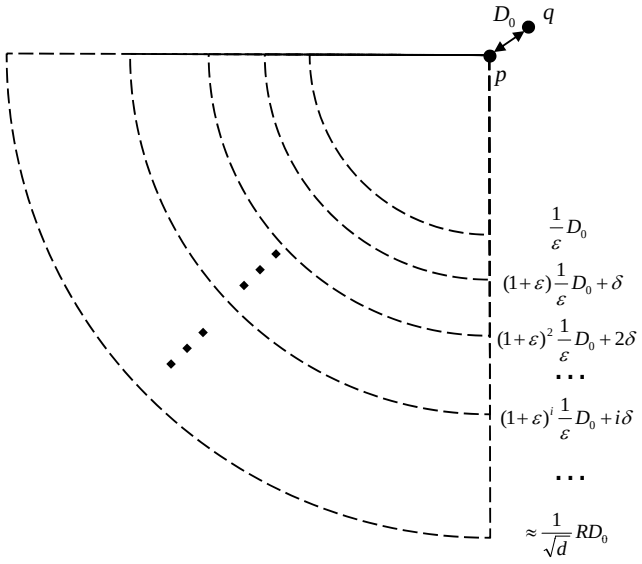


Figure 3.18 Example Virtual Zones in Two-Dimension

3.3.5 Conclusions

In this research, we addressed several ϵ-approximate geometric estimation problems in data streams over sliding windows. We proposed a diameter approximate algorithm which only uses $O((\frac{1}{\epsilon})^{\frac{d+1}{2}}\log R)$ space, and the worst running time to process a new point is $O((\frac{1}{\epsilon})^{\frac{d-1}{2}})$. We proved that our diameter algorithm can be applied to estimate convex hull over sliding windows. Finally, we studied the skyline estimation problem over sliding windows, and proposed a novel algorithm which uses $O(\frac{1}{\epsilon^d}\log R)$ space. A lower bound of this problem is provided. In the future, we will continue to study the geometric estimation problems over sliding windows.

CHAPTER 4. RESEARCH IN ATTACK ATTRIBUTION PART II: TRACEBACK TECHNIQUES

4.1 Stepping Stone Attack Attribution

4.1.1 Introduction

The number of network based attacks is growing because attackers can very easily hide their identities, and thereby reduce the chance of being captured and punished. It has become difficult and complicated to discover the true identity of attackers when they relay their attacks through ***stepping stones*** (intermediary hosts). The attack flow from the origin of the attack may travel through a chain of stepping stones before it reaches the victim. It is difficult for the victim to learn anything about where the attack comes from except that she can see the attack traffic from the last hop of the stepping stone chains. Therefore, it is desirable to design effective and efficient stepping stone attack detection schemes to attribute the attackers.

Several approaches have been designed to detect stepping stone attacks [23, 39, 98, 100, 106, 107, 108, 113, 119]. If the contents in the connections in the network are plain-text (i.e. unencrypted), it may be used to trace back to the origin of the attacker. Staniford-Chen and Heberlein [98] used thumbprints which are short summaries of the contents of a connection. The thumbprints can be compared to determine whether two connections contain the same text and are therefore likely to be part of the same connection chain. Wang et al. [108] injected detectable watermarks into the unencrypted traffic echoed back to the attacker, so that the attack can be traced back. For encrypted stepping stone connections, content based approaches cannot work any more. Neither do packet-size based approaches, if all packets are padded to the same size. Zhang and Paxson [119] proposed the first timing-based method which

only uses the packets' arrival time information. After that, several timing-based approaches were proposed [23, 39, 100, 106, 107, 113]. However, the attacker still can conceal its identity by destroying the time correlation between stepping stone connections. None of the current methods can effectively defend against delay and chaff perturbations simultaneously.

In this research, we discuss two different scenarios when delay and chaff perturbations exist, and propose three schemes [118]. We provide the upper bounds on the number of packets required to confidently detect stepping stone connections from non-stepping stone connections with any given probability of false attribution. We compare our schemes with previous approaches, and experimental results show that our schemes are more effective in these two scenarios.

4.1.2 Problem Definition

To avoid detection, an attacker may attack the victim using an encrypted link through several stepping stones, where the encrypted attack packets may show different contents among stepping stones while being padded to the same size. This indicates that one cannot use packet contents or packet sizes to detect and identify the attacker.

Several approaches have been proposed which only use the timing information. However, the attacker may evade detection by perturbing the timing information. It may introduce random delay before each packet departs from a stepping stone, or insert superfluous packets as chaff into the original attack flow on a stepping stone. In this research, we mainly consider two scenarios on stepping stone attribution problem:

Scenario 1: Only delay perturbation is introduced and no chaff perturbation exists.

Scenario 2: Delay and chaff perturbations exist simultaneously.

Not only can the attacker introduce delay and chaff perturbations, but the network itself may produce such perturbations. When packets travel through the network, the propagation delay of these packets is unavoidable. In anonymous networks, it is common that the attack connection is captured somewhere with several other connections which cannot be differentiated from the attack flow. Then these normal connections can seem to be as chaff to the attack

Table 4.1 Previous Schemes' Assumptions

Scheme	1	2	3	4
ON/OFF	Yes	No	–	Yes
Deviation	No	No	–	Yes
IPD	No	No	–	Yes
Watermark	No	Yes	–	Yes
State-Space	Yes	No	–	Yes
Multiscale	Yes	Yes	Yes	Yes
Detect-Attacks & Detect-Attacks-Chaff	Yes	Yes	Yes	Yes

flow. Therefore, the delay and chaff perturbations introduced to the attack flow may have two sources: one is the attacker, and the other is the network itself. No matter where the delay and chaff perturbations come from, we propose our solutions under the following assumptions:

1) The skew between the clocks of hosts where packets are captured is known.[1]

2) The total delay must be in the range of $[0, \Delta)$, where Δ is the ***maximum probable delay***[2].

3) The chaff perturbation is independent with the original flow.

4) No original packet is dropped, which means that each original packet will appear in both sides of the stepping stone connection.

Table 4.1 shows previous schemes' assumptions together, where '–' means that these schemes do not consider the scenario of chaff perturbation.

4.1.3 Our Schemes

To simplify the description of our schemes, let flow A denote the flow which contains only the original packets, and flow B denote a flow captured in the network. Flow B may be unrelated to flow A, or a related flow which contains all the original packets with or without chaff perturbation. Considering flows A and B in a stepping stone connection chain, our knowledge of the directional information between A and B has three possibilities:

[1]The hosts' clocks are different from each other. For simplicity, we assume the skew between different clocks is known such that we can compare the packets' timestamps from different hosts.

[2]In fact, the delay should be in the range of $[\Delta_{min}, \Delta_{max})$, where Δ_{min} is the minimum probable delay and Δ_{max} is the maximum probable delay and $0 \leq \Delta_{min} < \Delta_{max} < \infty$. For instance, if the two flows are linked through a satellite, Δ_{min} may be more than 1 second. However, we can change the delay range to $[0, \Delta)$ by skewing one flow left or right by Δ_{min}, and setting $\Delta = \Delta_{max} - \Delta_{min}$. To simplify the analysis in this research, we use the delay range $[0, \Delta)$ instead of $[\Delta_{min}, \Delta_{max})$.

1) A can only be an upstream flow of B;

2) A can only be a downstream flow of B;

3) A can be either B' upstream or downstream flow.

Sometimes we know the directional information. For instance, if flow A is the attack flow received by the victim, then B can only be A's upstream flow if they are correlated. If flow A is the response flow sent by the victim, then B can only be the downstream flow of A if they are correlated. However, if we do not know much about the causality of links, we have to consider both possibilities of flows sequence. To simplify the description of our schemes, we suppose that we have enough information that flow A has to be B's upstream flow if they are in a stepping stone connection chain. We expand our schemes to the other two cases later in this section.

4.1.3.1 Scheme *S-I* (for Scenario 1)

According to the assumption that maximum probable delay is bounded, each original packet i in flow A with arrival time u_i must have a corresponding packet in flow B within $[u_i, u_i + \Delta)$. In our scheme, we use Δ' to estimate Δ, because Δ may not be known by our scheme, and Δ' should be no less than Δ.

Scheme Description:

First, we set Δ' using foreknowledge. For the first observed original packet in flow A with arrival time u_1, we select the first arrival packet after u_1 in flow B as its corresponding packet. After the first corresponding packet is determined, each following original packet's corresponding packet is the successor of the previous original packet's corresponding packet. We observe whether all original packets' corresponding packets are in their probable arrival time range. If so, report CORRELATED and terminate. Otherwise, we repeat the observation with deferring each original packet's corresponding packet to the next one until we report CORRELATED, or until the arrival time difference between an original packet and its corresponding packet is larger than Δ' and we report UNCORRELATED.

In this scenario, if $\Delta' \geq \Delta$, the false negative rate of scheme *S-I* is 0%. Now we provide

a tight bound on the number of original packets that are required to be observed to achieve any given bounded false positives. Blum et al. [23] provide the upper bounds of their schemes. They first analyze the bounds by assuming that any normal flow can be modeled as a Poisson process with fixed Poisson rate. They then relax this assumption by assuming that any normal process can be modeled as a sequence of Poisson processes with varying rates and over varying time periods. In our analysis, we follow their steps and derive the expression of a tight bound. We must point out that the importance of the bound is not on the detailed value, but on the limits of the ability of attackers to evade detection by simply introducing delay and chaff perturbations.

We first assume all connections behave as Poisson processes and then generalize the assumption as Blum et al. do. Suppose original flow A and the other flow B are two unrelated flows with fixed Poisson rate λ_A and λ_B respectively. We begin to observe from time 0 and observe n packets in original flow A. Let $u_1, u_2, \cdots, u_n$ denote each original packet's arrival time in flow A. Then the probability function of these n packets arrival times is:

$$f(u_1, \cdots u_n) =$$

$$\begin{cases} \lambda_A^n e^{-\lambda_A u_n}, & \text{if } u_n \geq \cdots \geq u_1 \geq 0 \quad (4.1a) \\ 0, & \text{otherwise.} \quad (4.1b) \end{cases}$$

Let v_1 be the first packet's arrival time after u_1 in flow B, and $v_2, v_3, \cdots, v_m$ be the following packets' arrival times. Then the conditional probability function of these m packets arrival time on $u_1, \cdots, u_n$ is:

$$f(v_1, \cdots, v_m | u_1, \cdots, u_n) =$$

$$\begin{cases} \lambda_B^m e^{-\lambda_B (v_m - u_1)}, & \text{if } v_m \geq \cdots \geq v_1 \geq u_1 \geq 0 \quad (4.2a) \\ 0, & \text{otherwise.} \quad (4.2b) \end{cases}$$

Therefore we can obtain the joint probability function of $u_1, \cdots, u_n, v_1, \cdots, v_m$:

$$\begin{aligned} & f(u_1, \cdots, u_n, v_1, \cdots, v_m) \\ & = f(v_1, \cdots, v_m | u_1, \cdots, u_n) f(u_1, \cdots, u_n). \end{aligned} \quad (4.3)$$

Let Z_n denote the event that an unrelated flow is reported correlated by our scheme when we observe n original packets. Therefore the event Z_n's probability $P(Z_n)$ is equal to the false positive rate, and

$$\begin{aligned} P(Z_n) &= \int_{S_n^{(1)} \cup \cdots \cup S_n^{(\infty)}} f(u_1, \cdots, u_n, v_1, \cdots, v_\infty) \\ &\quad dv_\infty \cdots dv_1 du_n \cdots du_1 \qquad (4.4) \\ &\leq \int_{S_n^{(1)} \cup \cdots \cup S_n^{(n)}} f(u_1, \cdots, u_n, v_1, \cdots, v_{2n-1}) \\ &\quad dv_{2n-1} \cdots dv_1 du_n \cdots du_1 \qquad (4.5) \\ &\quad + (1 - \sum_{k=0}^{n} \frac{(\lambda_B \Delta')^k}{k!} e^{-\lambda_B \Delta'}), \end{aligned}$$

where

$$S_n^{(i)} = \begin{bmatrix} 0 \leq u_1 \leq \cdots \leq u_n < \infty \\ u_1 \leq v_1 \leq \cdots \leq v_{2n-1} < \infty \\ u_1 \leq v_i \leq u_1 + \Delta' \\ \cdots \\ u_n \leq v_{n+i} \leq u_n + \Delta' \end{bmatrix} \qquad (4.6)$$

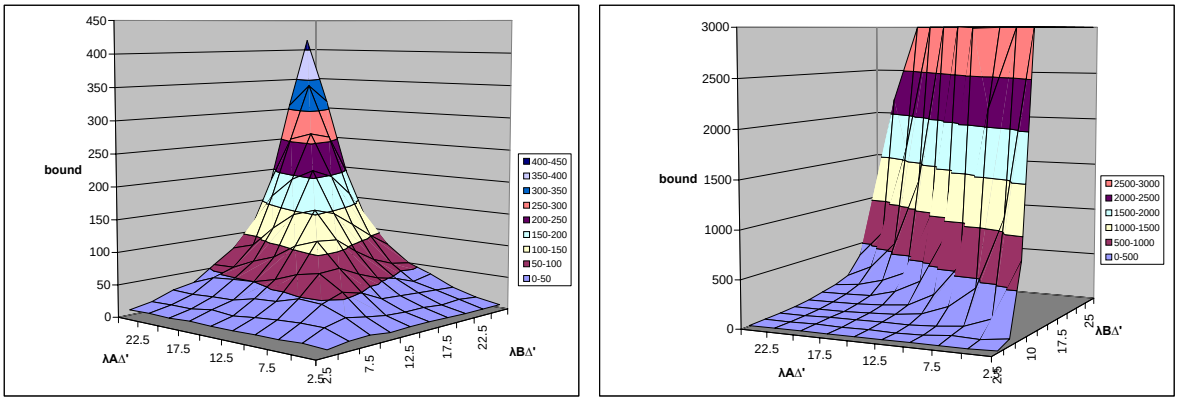
is the integral field when the ith packet after u_1 in flow B is selected as the first original packet's corresponding packet.

It is clear that the false positive rate decreases as the number of original packets we observe increases. For any given false positives, we may calculate the bound of the needed number of packets. Furthermore, we can prove that the bound of *S-I* is tighter than that of *Detect-Attacks*[23], which means *S-I* needs less packets than *Detect-Attacks* to achieve the same false positive rate.

Although it is difficult to calculate the bound using above formulas, we simulate *S-I* over millions of Poisson flows and obtain a computer simulated bound shown in Figure 4.1(a) when false positive rate is set to 1%. We make the following observations:

- For a given λ_A and Δ', the maximum bound occurs when $\lambda_B \approx \lambda_A$.
- For a given λ_A and λ_B, the bound increases quickly with the increase of Δ'.

- When $\lambda_A \Delta' \leq 7.5$, using less than 50 packets can achieve 1% false positive rate.
- Even when $\lambda_A \Delta' = 25$, using less than 450 packets can achieve 1% false positive rate.



(a) Bound of *S-I* when False Positive Rate is set to 1% (b) Bound of *S-II* when False Positive Rate is set to 1%

Figure 4.1 Bounds of *S-I* and *S-II*

4.1.3.2 Scheme *S-II* (for Scenario 2)

Scheme Description:

First, we set Δ' using foreknowledge. For each original packet i in flow A with arrival time t_i, we select the first arrival packet in the range $[t_i, t_i + \Delta')$ in flow B as its corresponding packet. If the packet has been selected by the previous original packet, select the first unselected packet. If we cannot select a corresponding packet in flow B for an original packet in flow A, we report UNCORRELATED. Otherwise, we report CORRELATED.

Let v_i be the arrival time of the ith original packet's corresponding packet in flow B. Then v_1 only depends on u_1, and v_i only depends on v_{i-1} and u_i when $i \geq 1$, so

$$f(v_1|u_1) = \begin{cases} \lambda_B e^{-\lambda_B(v_1-u_1)}, & \text{if } v_1 \geq u_1 \geq 0 \quad (4.7a) \\ 0, & \text{otherwise,} \quad (4.7b) \end{cases}$$

and

$$f(v_i|u_i, v_{i-1}) =$$

$$\begin{cases} \lambda_B e^{-\lambda_B(v_i-u_i)}, & \text{if } v_i \geq u_i \geq v_{i-1} \quad (4.8a) \\ \lambda_B e^{-\lambda_B(v_i-v_{i-1})}, & \text{if } v_i > v_{i-1} > u_i \quad (4.8b) \\ 0, & \text{otherwise.} \quad (4.8c) \end{cases}$$

We then obtain the joint probability function of $u_1, \cdots, u_n, v_1, \cdots, v_n$:

$$\begin{aligned} & f(u_1, \cdots, u_n, v_1, \cdots, v_n) \\ = \; & f(v_1, \cdots, v_n | u_1, \cdots, u_n) f(u_1, \cdots, u_n) \quad (4.9) \\ = \; & f(v_n | u_n, v_{n-1}) f(v_{n-1} | u_{n-1}, v_{n-2}) \cdots f(v_2 | u_2, v_1) \\ & f(v_1 | u_1) f(u_1, \cdots, u_n). \quad (4.10) \end{aligned}$$

Let Z_n denote the event that an unrelated flow is reported correlated by our scheme when we observe n original packets. Therefore, the event Z_n's probability $P(Z_n)$ is equal to the false positive rate, and

$$P(Z_n) = \int_{S_n} f(u_1, \cdots, u_n, v_1, \cdots, v_n) dv_n \cdots dv_1 du_n \cdots du_1, \quad (4.11)$$

where

$$S_n = \begin{bmatrix} 0 \leq u_1 \leq \cdots \leq u_n < \infty \\ v_1 \leq \cdots \leq v_n < \infty \\ u_1 \leq v_1 \leq u_1 + \Delta' \\ \cdots \\ u_n \leq v_n \leq u_n + \Delta' \end{bmatrix} \quad (4.12)$$

is the integral field.

Besides this tight bound on the number of original packets needed, we may also provide a loose bound:

Theorem 11.

$$P(Z_n) \leq (1 - e^{-\lambda_B \Delta'})^n, for\ n \geq 1. \quad (4.13)$$

Proof. We use mathematical induction to prove this theorem. First,

$$P(Z_1) = 1 - e^{-\lambda_B \Delta'}. \tag{4.14}$$

Suppose that for $i \geq 1$,

$$P(Z_i) \leq (1 - e^{-\lambda_B \Delta'})^i, \tag{4.15}$$

then

$$\begin{aligned} P(Z_{i+1}) &\leq P(Z_i) \cdot P(\text{at least one packet appears within} \\ &\quad [u_{i+1}, u_{i+1} + \Delta') \text{ in flow B} | Z_i) && (4.16) \\ &\leq (1 - e^{-\lambda_B \Delta'})^i (1 - e^{-\lambda_B \Delta'}) && (4.17) \\ &= (1 - e^{-\lambda_B \Delta'})^{i+1}. && (4.18) \end{aligned}$$

□

When we observe $\log_{1-e^{-\lambda_B \Delta'}} \delta$ packets, the false positive rate is at most δ.

We simulate *S-II* on millions of Poisson flows generated by computer and get a computer simulated bound shown in Figure 4.1(b) when false positive rate is set to 1%. We make the following observations:

- For a given λ_A and Δ', the bound increases quickly with the increase of λ_B.
- For a given λ_A and λ_B, the bound increases quickly with the increase of Δ'.
- For a given λ_B and Δ', the bound decreases with the increase of λ_A.

4.1.3.3 Scheme *S-III* (for Scenario 2)

Though the scheme *S-II* is simple enough, it requires that $\Delta' \geq \Delta$. If our foreknowledge cannot tell us what Δ should be, we have to set Δ' to a large 'safe' value, for instance, 100 seconds. However, according to our analysis on the bound of number of needed packets, as Δ' increases, the needed number of packets increases quickly, which means that for a given number of packets, the false positives rise with the increase of Δ'.

Here we propose a scheme to reduce false positives when the number of original packets is limited and Δ' is large. According to the probable delay bound, each packet of the original flow has a limited number of probable candidate packets in the other flow. That is, for an original packet i with arrival time u_i, only packets whose arrival times are in the range $[u_i, u_i + \Delta)$ on the other flow have the possibility to be the corresponding packet. Since the number of the original packets in flow A is limited, and each packet's number of candidates in flow B is also limited, we may reconstruct a limited number of probable flows from flow B with the same number of packets as the original flow A. The real corresponding flow of the reconstructed ones must be one of them. However, our goal is not to try and find the real corresponding flow, but to conclude whether the two flows are correlated or not. We then calculate the correlation of each probable flow with the original flow A using a certain correlation criterion, and use the largest correlation value to make the final decision.

Scheme Description:

First, we use Scheme S-II on the two flows and get a report. If it is reported uncorrelated, we return UNCORRELATED. Otherwise, we find all probable corresponding flows and choose a certain criterion to compute the correlation value between these two flows. We then make a decision.

This scheme is quite time consuming. Let n denote the total packets number in original flow A, and n_i denote the number of candidate packets for each original packet i. Let N be the total number of probable corresponding flows. Then

$$N \approx \prod_{i=1}^{n} n_i, \tag{4.19}$$

which is unacceptable in most cases. Therefore, for certain criteria, we hope to find some fast solutions.

We find that when we choose deviation as the criterion, we may construct a fast solution which reduces the number of probable corresponding flows required to calculate correlation to no more than $\sum_{i=1}^{n} n_i$. Let $v_i^{(j)}$ denote the ith corresponding packet's arrival time in the jth probable corresponding flow. Then the deviation between flow A and the probable

corresponding flow j is defined by[3]

$$dev^{(j)} = \frac{1}{n}\sum_{i=1}^{n}(v_i^{(j)} - u_i) - \min_{1\le i\le n}(v_i^{(j)} - u_i), \tag{4.20}$$

and the deviation between flow A and B is defined by

$$dev = \min_{1\le j\le N}(dev^{(j)}). \tag{4.21}$$

Fast Solution:

1 For each original packet i in flow A with arrival time u_i, we select the first arrival packet in the range $[u_i, u_i + \Delta')$ in flow B as its corresponding packet. If the packet has been selected by the previous original packet, then select the first unselected packet. This obtains the initial probable corresponding flow.

2 We search the original packet which has the minimum arrival time difference from its corresponding packet. This original packet updates its corresponding packet and selects the next packet in the flow. If the new corresponding packet to be selected has been selected by the next original packet, the next original packet in the flow needs to update its corresponding packet to the next one, and so on. We then obtain a new probable corresponding flow.

3 Repeat Step 2 until one of the original packets cannot select a corresponding packet.

Using the fast solution, we obtain a series of probable corresponding flows which are about $\sum_{i=1}^{n} n_i$. Furthermore, because each selected probable corresponding flow differs from the previous one with only 1 or several packets, then $dev^{(j)}$ can be derived from $dev^{(j-1)}$ easily. It can be proved that this fast solution achieves the same performance as the original scheme for the deviation criterion.

Theorem 12. *If deviation is chosen to be the correlation criterion, the fast solution achieves the same performance as the original scheme.*

Proof. Let $u_1, u_2, \cdots, u_n$ be the original packets arrival time of flow A, $w_1, w_2, \cdots, w_m$ be the packets arrival time of flow B, and $v_1, v_2, \cdots, v_n$ be the packets' arrival time of the corresponding flow which achieves the minimum deviation. According to the deviation definition, if the minimum gap $\min_{1\le i\le n}(v_i - u_i) = w_k - u_j$, which means that the minimum gap is achieved by the pair of original and corresponding packet (u_j, w_k), then the corresponding flow with

[3] We provide the definition of deviation here because it is slightly different with that in [113].

the minimum deviation must exactly be the flow that each original packet i chooses the first appeared packet in the range $[u_i+w_k-u_j, u_i+\Delta)$ which has not been chosen by its predecessor. The fast solution traverses all the probable combination of the pair of packets (u_j, w_k) with minimum gap, so it really achieves the corresponding flow with the minimum deviation. □

4.1.3.4 Complexity Analysis

Suppose we observe n packets in flow A and m packets in flow B. In the worst case of scheme *S-I*, the first packet in A has up to m candidate packets, so we must check such many rounds. In each round, we must check up to n packet pairs. Therefore the time complexity in the worst case is $O(n \cdot m)$. For fast solution of *S-III*, directly from what we discussed before, the time complexity is also $O(n \cdot m)$. However, both schemes normally have much better performance than their worst cases. For *S-II*, we need to select the corresponding packets among m packets in flow B, and each packet in B only needs to be checked once. Therefore the time complexity in the worst case is $O(m)$.

4.1.3.5 Directional Information Discussion

In the above description of our schemes, we assume that we know the directional information of case 1 that the original flow A has to be B's upstream flow if they are in a stepping stone connection chain. It is very simple to expand our schemes to the other two cases. In case 2 that we can confirm A is a downstream flow of B, we just simply add Δ' to the arrival time of each packet in flow B, and then execute the same schemes. In case 3 that we cannot confirm the direction between A and B, we simply run the schemes two times, one with directional information of case 1, the other with directional information of case 2. If both of the results are *UNCORRELATED*, we report *UNCORRELATED*. Otherwise, we report *CORRELATED*.

Table 4.2 Parameters Set

Scheme	Parameter	Value
ON/OFF	T_{idle}	0.5s
	δ	80ms
	γ	0.3
	γ'	0.02
	$\min_{csc}$	2
Deviation	Threshold	6s
IPD	s	20
	δ_{cp}	0.9
	δ	0.6
State-Space	weight updating rate	0.1
	Threshold	0.5
Multiscale	ψ	bump
	s	$64s$
	Threshold	0.7
Detect-Attacks & ***Detect-Attacks-Chaff***	Δ	20s
	δ	0.01
S-I, S-II & ***S-III***	Δ'	30s
S-III	Threshold	6s

4.1.4 Experimental Evaluation

To make comparison, we implement the following six approaches: *ON/OFF* [119], *Deviation* [113], *IPD* [107], *State-Space* [100], *Multiscale* [39] and *Detect-Attacks/Detect-Attacks-Chaff*[4] [23]. We do not compare these passive schemes with *Watermark* which is the only active approach. We compare these six present schemes with scheme *S-I* in Scenario 1, and with schemes *S-II* and *S-III* in Scenario 2.

We list the values of all used parameters of these nine approaches in Table 4.2. These values are the original values used by the authors or typical values if the authors do not clearly indicate. We omit the meaning of these parameters which may be found in their original papers. Table 4.2 also shows that *S-I* and *S-II* have the smallest number of control parameters, so they can be easily implemented. In contrast, *ON/OFF* has five control parameters, and it is hard to train and test whether they are set to the optimum values.

In our experiments, we use the data set of Auckland-IV traces in NLANR PMA Daily Traces Archive [2]. We extract 293 Telnet/SSH flows which have more than 1000 packets,

[4]Detect-Attacks is used in experiments for Scenario 1, and Detect-Attacks-Chaff is used in experiments for Scenario 2.

and totally 1.3 million packets. These flows' packet rate distribution is shown in Figure 4.2. However, most of these flows are not overlapped, and therefore any scheme with Assumption 1 can get high score, which is unfair to other schemes. Therefore, in our experiments, we randomly adjust the flows' start time such that they always overlap. Furthermore, in all our experiments, we only consider such pairs of test flows that the one seemed as original flow A is totally covered by the other one. We have also built a stepping stone attack attribution test-bed environment consisting of 40 machines and developed an automated interactive traffic (Telnet, SSH, etc.) generator [111]. We modified telnetd and sshd source code so that they can introduce random delay and chaff perturbations with various controllable distributions into stepping stone traffic.

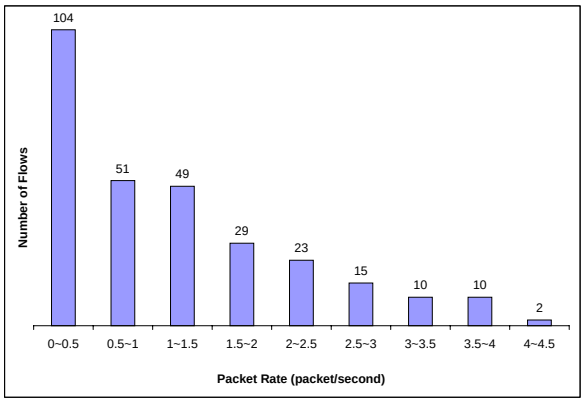


Figure 4.2 Packet Rate Distribution

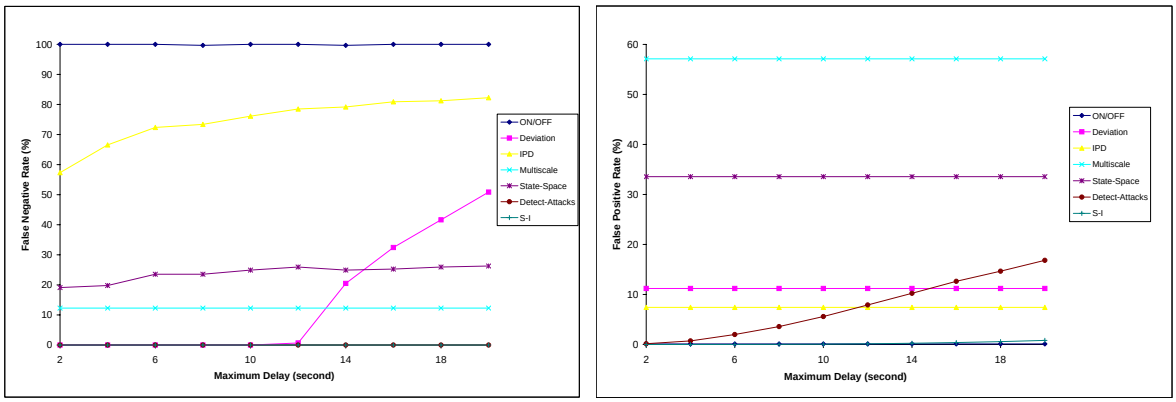
4.1.4.1 Experiments Set 1

This set of experiments is designed to answer the following questions:

1) How effectively can different schemes detect stepping stones in Scenario 1 with different delay perturbations added?

2) How effectively can different schemes detect stepping stones in Scenario 2 with different chaff perturbations added?

For scenario 1, we add uniform distributed delay to each original flow and get 293 delayed flows. Then we calculate the correlation between each original flow and each delayed flow.



(a) False Negative Rate (b) False Positive Rate

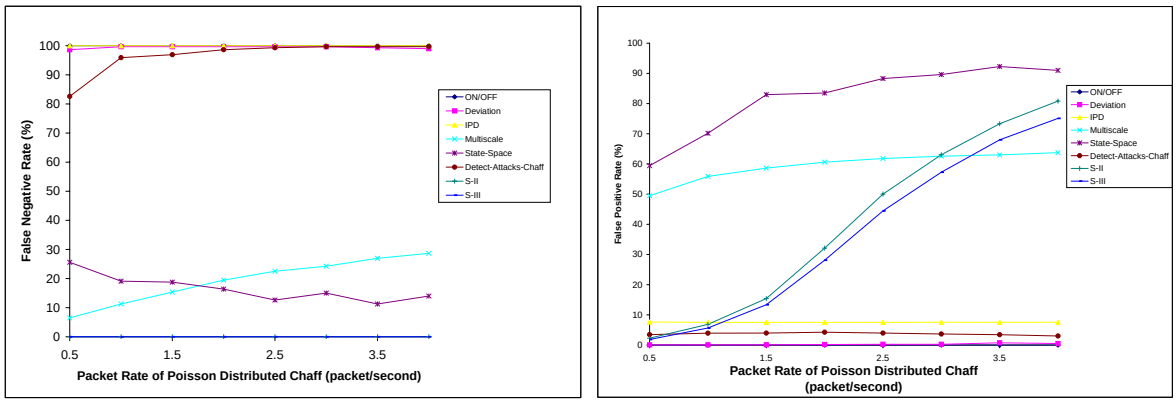
Figure 4.3 Scenario 1 with Different Delay Perturbations

Thus we get 293 stepping stone connection pairs and 66,140 non-stepping stone connection pairs. We consider 10 different kinds of uniform delays, and their maximum delays increase from 2 seconds to 20 seconds with a step of 2 seconds. To avoid the influence of different packets number of flows, each scheme must return its result after they receive 250 original packets. Figure 4.3 shows the false negative rates and false positive rates of these seven approaches. We make the following observations:

- Only *Detect-Attacks* and *S-I* can detect all stepping stone attacks. However, the false positive rate of Detect-Attacks is much higher than that of *S-I*. The reason is that *S-I* has a tighter bound than *Detect-Attacks*.
- Using only 250 packets, *S-I* can achieve as low as 0.82% false positive rate even when the maximum delay is 20 seconds.

To simulate Scenario 2 that delay and chaff perturbations exist simultaneously, we first add uniform distributed delay with maximum of 10 seconds. We then generate Poisson distributed chaff with packet rate 0.5 packet/s, 1 packet/s, ..., 4 packet/s. For each scheme, 1000 original packets are used. Figure 4.4 shows the false negative rates and false positive rates of these eight approaches. We make the following observations:

- Only *S-II* and *S-III* can detect all stepping stone attacks. *S-III* is a bit better than *S-II*.
- *ON/OFF* and *Deviation* always report there is no attack.



(a) False Negative Rate (b) False Positive Rate

Figure 4.4 Scenario 2 with Different Chaff

- *Detect-Attacks-Chaff* totally loses detecting ability. The reason is that the added chaff perturbation exceeds its detection bound.
- The false positive rates of *S-II* and *S-III* increase with the packet rate of chaff, which complies our previous analysis. They need more original packets to achieve arbitrary low false positives when chaff perturbation is heavy.

4.1.4.2 Experiments Set 2

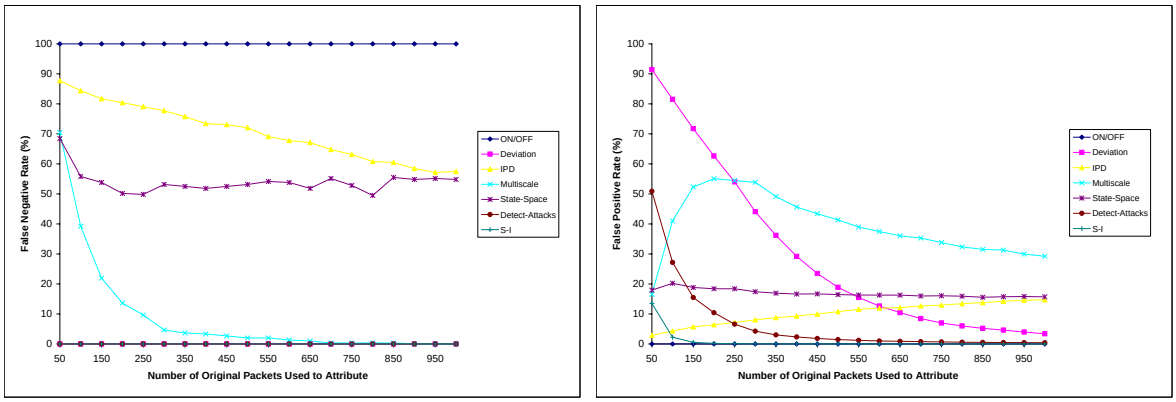
This set of experiments is designed to answer the following questions:

1) How effectively can different schemes detect stepping stones in Scenario 1 with different number of available original packets?

2) How effectively can different schemes detect stepping stones in Scenario 2 with different number of available original packets?

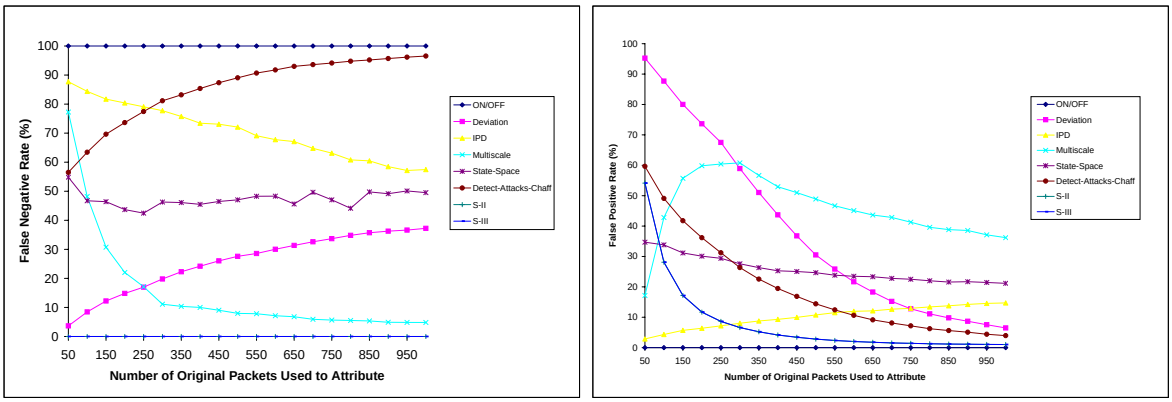
For scenario 1, we add uniform distributed delay with maximum of 10 seconds. The number of available packets increases from 50 to 1000. Figure 4.5 shows the false negative rates and false positive rates of these seven schemes. We make the following observations:

- Deviation, Detect-Attacks and *S-I* can detect all stepping stone attacks. However, the false positive rates of Deviation and Detect-Attacks are higher than that of *S-I*.
- Using only 150 packets, *S-I* can achieve as low as 0.54% false positive rate.



(a) False Negative Rate (b) False Positive Rate

Figure 4.5 Scenario 1 with Different Number of Original Packets



(a) False Negative Rate (b) False Positive Rate

Figure 4.6 Scenario 2 with Different Number of Original Packets

To simulate Scenario 2 that delay and chaff perturbations exist simultaneously, we use the real flows as the source of chaff perturbation. We chose the longest 30 flows from these 293 original flows. We combine one of the residual 263 flows with one of these 30 flows together and get $263 \times 30 = 7,890$ combined flows. We then add uniform distributed delay with maximum of 10 seconds to these combined flows. We calculate the correlation between each original flow and each combined flow. In these combined flows, the packets from one of the 30 flows may be seemed as chaff perturbation to the other flow. We finally get $7,890$ stepping stone connection pairs and $1,649,870$ non-stepping stone connection pairs. Figure 4.6 show the false negative rates and false positive rates of these eight approaches. We make the following observations:

- Only *S-II* and *S-III* can detect all stepping stone attacks. The false positive rates of *S-II* and *S-III* totally overlap in Figure 4.6(b). It means that when used in real flows, *S-II* is good enough and we need not to use *S-III* which is more time consuming.
- Detect-Attacks-Chaff totally loses detecting ability.
- Multiscale can achieve 10% false negative rate when using more than 500 packets. However, its false positive rate is about 40%.
- *S-II* and *S-III* can achieve about 1% false positive rate using 1000 packets.

4.1.5 Conclusion

Network based attackers often launch attacks through stepping stones to evade detection, who also make detection even more difficult by encrypting attack traffic and introducing delay and chaff perturbations. Several approaches have been proposed to detect stepping stone attacks. However, none of them performs effectively when delay and chaff perturbations exist simultaneously.

In this research, we discuss two different stepping stones scenarios and proposed three schemes to attribute stepping stone attacks. Two of the three schemes can effectively detect stepping stones even when delay and chaff perturbations exist simultaneously. We analyze our schemes and provide the bounds on the number of packets needed to confidently detect stepping stone connections from non-stepping stone connections with any given probability of

false attribution. We compare our schemes with previous ones and experiments show that our schemes are more effective in detecting stepping stones in the two scenarios.

Our mathematical analysis and experimental results show that although the attackers can make detection more complicated by introducing delay and chaff perturbations, if the attack connections are long enough, they cannot evade detection only by introducing limited and independent delay and chaff perturbations. Attackers may be forced to introduce dependent delay and chaff perturbations, and reduce the number of attack packets. As a next step, we are continuing the research on attributing stepping stone attacks with message merge and split [112].

4.2 Topology-aware Single Packet Attack Traceback

4.2.1 Introduction

With the phenomenal growth of the Internet, more and more people enjoy and depend on the convenience of its provided services. Unfortunately, the number of network-based attacks is also increasing very quickly. The consequences are serious and, increasingly, financial disastrous, but ironically, only few of the attackers have been captured and thereby punished because of the stateless nature of the Internet. Network-based attackers can easily hide their identities through IP spoofing, stepping stones, network address translators (NATs), Mobile IP or other ways, and thereby reduce the chance of being captured. The current IP network infrastructure lacks measures which can effectively deter and identify motivated and well-equipped attackers. As a result, due to the lack of effective attack attribution techniques and concerns for negative publicity, the percentage of organizations reporting computer intrusions to law enforcement has continued its multi-year decline [52]. Therefore, it is desirable to design effective and efficient IP traceback systems to attribute attackers and help to reconstruct cyber crime scenes.

Building systems that can reliably trace attack packets back to their real origins in the current IP networks is a first and important step in making cyber criminals accountable. There are many forms of network-based attacks. While the most-widely reported DDoS attacks

are conducted by flooding networks with large amounts of traffic, there are network-based attacks that require significantly smaller packet flows. Some attacks (e.g., Teardrop) can even succeed by using only one well-targeted packet. In building such traceback systems, there are a number of significant challenges including which packets to trace, how to trace *long-lifetime*[5] and *short-lifetime*[6] attacks, and how to minimize router processing overhead and storage space requirements, while satisfying applications' and network users' privacy requirements.

Several IP traceback schemes have been designed to attribute the origin of IP packets through the Internet. We roughly categorize them into four primary classes: (i) *Active Probing* [25, 99], (ii) *ICMP Traceback* [21, 71, 110], (iii) *Packet Marking* (including deterministic, probabilistic, and algebraic packet marking) [20, 36, 85, 93, 96], and d) *Log-based Traceback* [66, 74, 94, 95]. The IP traceback systems built atop of approaches of Active Probing, ICMP traceback and Packet Marking usually require a sufficiently large set of attack packets to make traceback possible, which are not effective to traceback short-lifetime or single packet attacks. Log-based traceback schemes seem suitable to attribute individual packet to its origin. However, several log-based methods such as [74] require recording the entire network traffic for future attack traceback. Obviously, such methods have overly high storage space requirements, which make them impractical to be used for current high-speed IP networks, especially those with heavy traffic.

Thereafter, to address the problem of log-based IP traceback systems' overly large storage space requirement, two IP traceback systems [94, 95] were designed using Bloom filters [22], a space-efficient data structure for representing a set of elements to respond membership queries. However, although Bloom filters are space-efficient data structures in responding membership queries, they have inherent unavoidable collisions which produce false positives, and thus restrain the effectiveness of these systems.

In this research, we propose a Bloom filter-based topology-aware single packet IP traceback system, namely *TOPO*, which utilizes router's local topology information, i.e., its immediate

[5]By *long-lifetime*, we mean that there are a sufficiently large number of attack packets available for traceback systems.

[6]By *short-lifetime*, we mean that there are a significantly smaller number of attack packets available for traceback systems. Some attacks might only use a single packet.

predecessor information, to traceback [114]. Our theoretical analysis and experimental evaluation show that TOPO can significantly reduce the number and scope of unnecessary queries and thus, significantly decrease the false attributions to innocent nodes.

Furthermore, we consider the practicability of Bloom filter-based IP traceback systems. In some real world networks, it is difficult or even impossible to install an IP traceback system on all the routers on the network. Therefore, partial deployment is an important and desired property when designing and implementing IP traceback systems. We analyze and show that TOPO is suitable to be partially deployed while maintaining its traceback capability.

Finally, we discuss an issue on utilizing Bloom filters which has never been studied. When Bloom filters are applied in IP traceback systems, it is difficult to decide their optimal control parameters *a priori* and thus achieve the lowest false positive rate. Based on our analytical results, we design a k-adaptive mechanism to dynamically adjust parameters of Bloom filters such that our IP traceback system can achieve the best performance in terms of false attribution rates and storage space requirement.

4.2.2 Problems and Goals

4.2.2.1 Problems in Existing Traceback Schemes

To prevent the Internet from being attacked, it is desirable to design effective and efficient IP traceback systems to attribute the attackers and help reconstruct cyber crime scenes. *IP traceback problem* is to traceback the network path(s) the attack traverses and identify the real attackers.

Active Probing, ICMP traceback and packet marking schemes are designed to traceback those long-lifetime attacks, and they are not suitable to traceback single packet attacks or short-lifetime attacks. SPIE is designed to trace the origin of a single IP packet delivered by the network in the recent past. In SPIE, when an intrusion detection system (IDS) detects an attack, it sends out a query message to SPIE. SPIE then dispatches the query message to the relevant routers for processing. After a router receives a query about whether it has forwarded a packet with a specific arrival time, it checks its Bloom filter for that time interval. If the result

is 'present' in the Bloom filter, the router must continue to query all its upstream neighbors. Consequently, there would be a lot of unnecessary queries sent to innocent routers. The unnecessary queries force the routers to spend CPU time and other resources to respond, and thus reduce the performance of their tasks as routers. Furthermore, these unnecessary queries can introduce innocent nodes into the attack graph because of the unavoidable false positives of Bloom filters, which in turn cause false attributions. Consequently, it is desirable to design new mechanisms to control the unnecessary queries and thus reduce the false attributions to innocent nodes.

For any proposed IP traceback systems, besides traceback effectiveness, practicability is also an important and desired property in evaluating their system performance. In some real world networks, it is difficult or even impossible to install Bloom filter-based IP traceback systems, such as SPIE [95], PAS [94] and TOPO proposed in this research, on all routers. Although some routers in these networks may easily install and activate a Bloom filter-based IP traceback system, there are some routers such as core routers that are hardly to be updated because of the high overhead involved or other administrative issues.

If a Bloom filter-based IP traceback system can be partially deployed without losing (or compromising a little) traceback capability compared with the fully deployed system, then it will have the following potential advantages:

- Low cost: Such a system may avoid wasting money and labor on updating routers which are difficult and expensive to update. In addition, it is possible to reduce memory space required by the whole traceback system [69].

- Flexibility: Such a system may be implemented on networks which have routers that are impossible to be updated. Furthermore, it also provides flexibility to the administrator who launches traceback or investigation of a particular attack. She may enable only a portion of network routers and thereby avoid alerting the attacker.

Therefore, an IP traceback system which can be practically deployed should have the property of partial deployment.

Bloom Filters have been used in many network applications. In some applications, the element number n that should be inserted is known *a priori* before the construction of Bloom filters. For instance, Bloom filters are used to store a dictionary of unsuitable passwords for security purposes, where the number of passwords is countable [97]. With the fixed element number n, given any required false positive rate f and the memory size m, the optimal hash function number k can be calculated by equation (2.2). However, in IP traceback systems, the traffic volume that needs to be recorded varies significantly, especially when the network is under attacks. This indicates that the packet number n is unknown *a priori*, when m and k have to be decided in advance before traceback system is deployed. For different values of n, there are different optimal values of k. Therefore, a Bloom filter-based IP traceback system faces the problem of how to adaptively adjust parameter k to achieve a better false positive rate.

4.2.2.2 System Model

We consider IP networks (the Internet) where TCP/IP architecture and protocols are being used. An IP network consists of a number of host computers and network devices (e.g., routers or switches), which are connected by physical links on which packets are forwarded. We make the following assumptions in IP networks:

- Most routers are reliable.
- Some routers' software and hardware can be updated.

4.2.2.3 Threat Model

With the phenomenal growth of Internet, more and more people enjoy and depend on the convenience of its provided service. Meanwhile, the number of network-based attacks is increasing very quickly. The reality is that only few of the attackers have been captured and punished due to the stateless nature of the Internet and the readily available tools and techniques on the Internet that are easily taken to conceal themselves from being discovered.

It is well-documented that many attackers use spoofed IP source addresses in their attack packets. As these packets traverse the Internet, little useful information is left for the victims to identify their true origins. Some attackers also launch their attacks behind a chain of compromised machines which are called "*stepping stones*". Some attacks, such as *SYN flood Denial-of-Service* (DoS) attack, need to flood network links with large amounts of packets, while there are other attacks which require significantly smaller packet flows. Furthermore, some sophisticated attackers can start and finish their attacks using a single well-targeted packet, such as *LAND* attack [3], *Ping of Death* attack [4] and *Teardrop* attack [5].

4.2.2.4 Problem Definition and Our Goals

In this research, we aim at designing a Bloom filter-based topology-aware single packet IP traceback system, namely TOPO, to solve the problems discussed above. We set the following as our goals in the design of TOPO:

1. To design a single packet IP traceback system which has fewer unnecessary query messages and fewer false attributions to innocent nodes.
2. To design a single packet IP traceback system which needs not to be fully deployed in the entire network.
3. To design a mechanism which helps achieve the best performance of Bloom filters by adaptively adjust parameter k.

4.2.3 System Description

In this subsection, we first introduce topology-aware IP traceback mechanism which is the baseline idea of TOPO, and then discuss the partial deployment issue and the design of the k-adaptive mechanism for dynamically adjusting parameters of Bloom filters to achieve better performance.

4.2.3.1 Topology-aware IP Traceback

We first define three terminologies used in this research:

- ***Packet Signature*** is defined as the information to identify individual packet from each other.
- ***Packet Predecessor*** is defined as the immediate upstream neighbor which sends the packet to the current router.
- ***Predecessor Identifier*** is defined as the information to identify the predecessor of a packet from other upstream neighbors.

As we discussed above, we need to find a way to control the production of unnecessary queries. This can be achieved if the routers have the ability to identify which upstream routers should be queried and which else should not be queried. Therefore, if we can check not only the presence of each packet but also the packet predecessor information, we can only query the exact predecessor and thus significantly reduce the unnecessary queries. This requires that we record not only the packet signature but the predecessor information.

An intuitive way to record packet predecessor information is to separate the incoming packets into several Bloom filters, each of which only store the packets coming from a distinct predecessor. Snoeren et al. [95] discussed that a router may maintain separate Bloom filters for each of its input port, because different upstream neighbors typically use different input ports. Although this complicates the query process, the input port isolation may reduce the number of upstream neighbors that need to be queried. However, the number of predecessors (or active ports) within a certain time interval is an unknown parameter: Most routers cannot decide how many upstream neighbors will send packets to it within a time interval in a real world network. Even though we may estimate the maximum probable number of predecessors, another problem appears that how to divide the limited memory on the router for the Bloom filters of individual predecessors. We do not know how many packets will be received from a certain predecessor. As we learn from equation (2.1), the false positive rate of Bloom filters depends on m/n. If we divide the memory equally, then the Bloom filter which a large amount of packets are inserted into will have high false positive rate. Therefore, it is not an effective and practical solution to divide the limited memory into several separate Bloom filters.

Another way of recording predecessor information is to create a list for each bit which is set to 1 in the Bloom filter. This list is used to record all the predecessors which have set this bit. With these lists, the presence of a packet is that all the k corresponding bits are set to 1, and there is at least one common predecessor which appears on all the k corresponding lists. However, the extra lists will consume a lot of memory, and thus degrade the Bloom filter's performance or increase the system memory requirement.

We propose that the router still only maintains one Bloom filter at a time. The Bloom filter not only keeps the packet signature, but also the predecessor identifier. The predecessor identifier can be hashed into the Bloom filter with the packet signature together. Such an operation really can record the topology information without increasing false positive rate or requiring larger memory space.

Now we introduce TOPO which is based on the above idea. TOPO is constructed on some special routers which are equipped with Bloom filters, and we call them TOPO-equipped routers. When a packet travels through the network where TOPO is deployed, no matter whether it is an attack packet or not, the TOPO-equipped routers on its path record the packet signature and predecessor information. If an attack packet is identified by the victim, the victim's address, packet signature, and packet arrival time, are reported to TOPO as a traceback request. TOPO then begins traceback by sending a query message to the TOPO-equipped router responsible for the region containing the victim. This router then checks its record and continues to query other TOPO-equipped routers if necessary. Finally, all responses from queried TOPO-equipped routers are gathered by TOPO to generate the attack graph. The attack graph is used for further analysis and corresponding actions.

We show the details of TOPO-equipped router's behaviors when it records a packet and responds a query in Figure 4.7 and 4.8 respectively.[7]

Record a Packet

- When a TOPO-equipped router receives a packet, it first extracts the packet signature

[7]We do not discuss the topic of malicious behaviors of compromised routers, which is beyond the scope of this research. We also do not discuss the packet transformation issue, which is well discussed in [95].

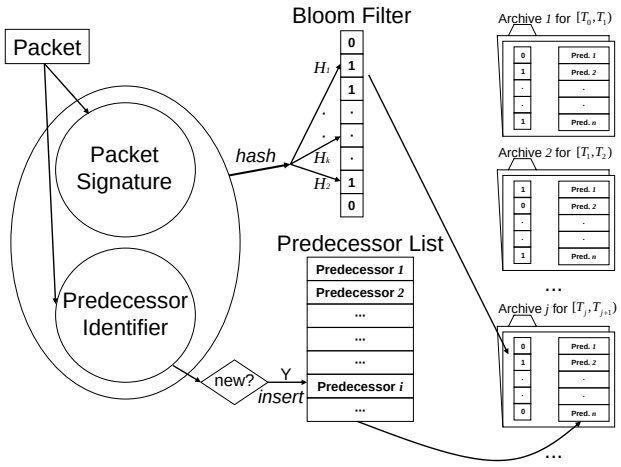


Figure 4.7 Router's Behaviors when Receiving a Packet

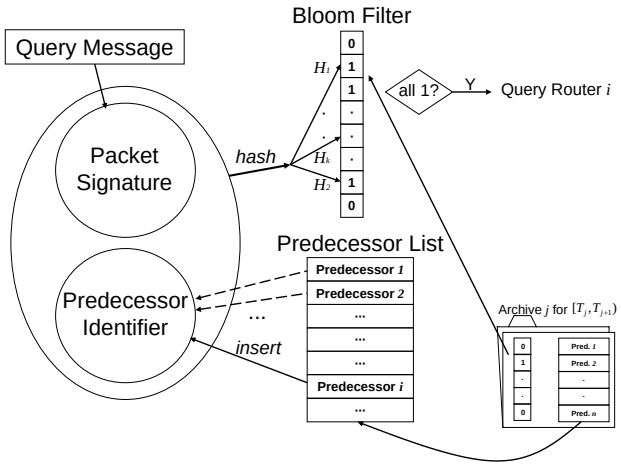


Figure 4.8 Router's Behaviors when Receiving a Query Message

and predecessor identifier.

- The predecessor identifier is inserted into an extra predecessor list if the predecessor has not been inserted before.
- The router calculates the k hash values of the combination of packet signature and predecessor identifier, and inserts them into its Bloom filter by setting the corresponding bits to 1.
- At the end of the anticipated time interval, the current Bloom filter and the predecessor list are archived by flushing the oldest ones, and a new Bloom filter and predecessor list start.

Respond a Query

- When a TOPO-equipped router receives a query message, it first retrieves the Bloom filter and the predecessor list for the relevant time interval using the given attack packet arrival time in the query message.
- Each predecessor identifier on the list is combined with the packet signature provided by the query message, and the combination is hashed using the same k hash functions to check if it is present in the Bloom filter.
- The router will respond the query that it forwarded this packet before if there is at least one presence found. If so, it only continues to query the predecessor(s) which is (are) present in the Bloom filter.

For a router which is not TOPO-equipped, when receiving a packet, it just simply forwards it to the next hop in the path; when receiving a query message, it simply forwards it to all of its upstream neighbors.

4.2.3.2 Partial Deployment

Ideally we can equip TOPO on all routers on the network, and thus can trace back single packet effectively. However, many legacy routers cannot be updated, or there is no enough

money to update all routers. Therefore, with resource and policy restriction, in most cases we have to partially deploy an IP trace back system. One simple partial deployment mechanism is that Bloom filter-based IP traceback system is only installed on edge routers and all internal routers remain unchanged. Under such architecture, all edge routers will be queried for any traceback request. Therefore all innocent end nodes have the possibility to be considered attackers, and thus introduce more false positives. Furthermore, there are two other drawbacks using such a partial deployment mechanism. First, the victim cannot get a full attack graph; second, the query burden on each edge router might be very heavy. Therefore, it is necessary to deploy Bloom filters at least on part of internal routers if the full attack graph is desired or traceback requests are frequently delivered.

To partially deploy TOPO, we need to find a way to select particular routers to equip TOPO to achieve the highest traceback performance with the lowest cost. We call it *TOPO-equipped routers placement problem*. Actually, this problem should be solved as an optimization problem. For instance, given the network map, the network traffic information, the distribution of attacks, and the costs of updating routers to TOPO-equipped, we optimize the traceback system with respect to the total cost and traceback performance (i.e., a combination of cost, false positives and false negatives). When the number of candidate routers is huge, it is time-consuming to find the optimized result. Furthermore, it is hard to get the distribution of attacks before deployment of TOPO.

In this research, we propose an intuitive but effective method to place the TOPO-equipped routers. The basic idea is that if the distribution of attacks is not available, intuitively an evenly distributed IP traceback system should have better performance. We assume that we know all the possible paths of the network, which can be achieved through some known network mapping tools [31, 53, 58]. To evenly distribute N TOPO-equipped routers on the network, we first sort all paths in descending order. We then equip TOPO on the median router of the longest path. Each path through this router is divided into two shorter paths. We sort all paths again and choose the median router of the longest path. We repeat these steps until we equip TOPO on N routers. We provide the performance analysis of partially deployed TOPO

in the next subsection.

4.2.3.3 k-Adaptive Mechanism for Bloom Filters

When a new Bloom filter starts to record the arriving packets, the exact number of packets that would be inserted into this Bloom filter is unknown. As shown in formula (2.1), the false positive rate is decided by n and k when m is fixed. Although we can estimate a possible value of n using historical knowledge about network statistics and choose a corresponding k, in some scenarios n may dynamically change in a large range, especially when the network is under attacks. In such cases, the false positive rate will be (greatly) higher than the optimal rate, as shown in Figure 4 of [43]. However, we cannot reconstruct a Bloom filter with the optimal k after we finally know the packet number n, because when we notice that we should use a better k, the previous packets have passed because of the limited memory on routers. Therefore, we have no chance of selecting a better k and hashing all the previous packets again.

Someone may argue that when a Bloom filter is saturated, it can be archived and a new filter starts. However, when the memory is limited, there may not be extra memory space for those unexpected packets. For instance, a router in the IP traceback system is designed to trace the traffic within 1 hour with the granularity of 1 minute, and divides its memory into 61 slices (1 slice is used to store the packets in the current minute, and the other 60 slices are for these 60 archived Bloom filters in the recent past.). If the router finds that the current Bloom filter saturates after 30 seconds, it cannot archive the current Bloom filter by flushing the oldest archived filter, because such an operation makes the router fail to respond the queries of the previous packets between 59 minute 30 second and 60 minute ago, and thus violates the traceback system goals and requirements. Otherwise, after an attacker finishes a serious attack, it can easily flood the network and flush its previous attack traffic before the system is aware of its attack and starts the traceback. Therefore, we need a mechanism to adjust the hash function number k with respect to the dynamic n to achieve the best performance of Bloom filters.

A direct solution is to construct several Bloom filters simultaneously which have different

numbers of hash functions. After all elements are inserted, we archive the Bloom filter with the optimal k and throw away all others. Obviously, this solution requires much more memory for the extra Bloom filters, and thus perhaps decreases the entire performance eventually.

We propose an effective k-adaptive mechanism as shown in Figure 4.9 which uses a table v with m Q-bit entries to record the results of K m-bit Bloom filters with different numbers of hash functions, if every smaller hash function set is the subset of larger ones, where

$$Q = \lceil \log_2(K+1) \rceil. \tag{4.22}$$

Let $H_1, H_2, \cdots, H_K$ be the K hash function sets which have $k_1, k_2, \cdots, k_K$ hash functions respectively, and $k_1 < k_2 < \cdots < k_K$.

$$\begin{aligned} H_1 &= \{h_1, \ldots, h_{k_1}\}, \\ H_2 &= \{h_1, \ldots, h_{k_2}\}, \\ &\cdots \\ H_K &= \{h_1, \ldots, h_{k_K}\}. \end{aligned}$$

Therefore $H_1 \subset H_2 \subset \cdots \subset H_K$. For each hash function h_i among the total k_K hash functions, let s_i denote the number of hash sets it belongs to in $H_1, H_2, \cdots, H_K$. For instance, if $h_i \notin H_j$ and $h_i \in H_{j+1}$, then $s_i = K - j$. s_i is a number between 1 and K.

At the beginning, the whole table v is initialized to 0. When a packet arrives, the packet signature pkt and predecessor identifier p_{id} are extracted and hashed using each hash function h_i. s_i is written into entry $h_i(pkt, p_{id})$, if s_i is larger than the entry's current value. After current time t passes the required end time t_{end}, the optimal k_j is calculated among $k_1, k_2, \cdots, k_K$ which minimizes false positive rate f based on the actually inserted element number n using formula (2.1). The next job is to restore and archive the Bloom filter b with the optimal k_j hash functions. Each entry's value $v(i)$ is compared with $K - j$. If $v(i)$ is larger than $K - j$, the corresponding bit $b(i)$ in Bloom filter is set to 1.

We use the following example to demonstrate the k-adaptive mechanism in detail. Let

(1) Initialize the table v and Bloom filter b to 0

(2) $n \leftarrow 0$

(3) while ($t < t_{end}$ AND receive a new packet pkt with p_{id})

(4) $\quad n \leftarrow n + 1$;

(5) $\quad$ for ($i \leftarrow 1$ to k_K)

(6) $\quad\quad j \leftarrow h_i(pkt, p_{id})$

(7) $\quad\quad v(j) \leftarrow \max(v(j), s_i)$

(8) Find the index j of k_j which minimizes f in $k_1, k_2, \cdots, k_K$

(9) for ($i \leftarrow 1$ to m)

(10) $\quad$ if ($v(i) > K - j$)

(11) $\quad\quad b(i) \leftarrow 1$

Figure 4.9 k-adaptive Procedure

$m = 10^6$, $Q = 2$ and $K = 3$. Suppose we want to choose the best k among $k_1 = 1$, $k_2 = 3$ and $k_3 = 4$, and the hash function sets are $H_1 = \{h_1\}$, $H_2 = \{h_1, h_2, h_3\}$, $H_3 = \{h_1, h_2, h_3, h_4\}$. Instead of recording 3 m-bit Bloom filters b_1, b_2 and b_3 in the memory which use the hash set H_1, H_2, and H_3 respectively, we only keep a table v with m 2-bit entries. Now we show that we can restore b_1, b_2 and b_3 using v. We observed that if finally an entry $b_1(i)$ is set to 1, then the entries $b_2(i)$ and $b_3(i)$ must also be 1, because $H_1 \subset H_2 \subset H_3$. Therefore, there are only 4 possible value combinations of the 3 entries $b_1(i)$, $b_2(i)$ and $b_3(i)$. As shown in Table 4.3, we can find a bijection between $b_1(i)$, $b_2(i)$, $b_3(i)$ and the number of 1s in them, and thus we can use only 2 bits to represent the 3 entries. In this case, $s_1 = 3$, $s_2 = s_3 = 2$, and $s_4 = 1$, and they represent the number of 1s that the corresponding hash function will set in the 3 Bloom filters. If finally we receive $n = 2.5 * 10^5$ packets, we get the following false positive rates using formula (2.1): $f_{b_1} = 0.221$, $f_{b_2} = 0.147$, $f_{b_3} = 0.160$. Therefore, we choose $k = k_2 = 3$ and restore Bloom filter b_2. We compare each entry $v(i)$ with $K - 2 = 1$ and set $b(i)$ to 1 if $v(i)$ is larger.[8]

[8]When $K = 3$, we can effectively use logical operations instead of comparison operations to restore any desired Bloom filter from the table. b_1 can be restored by AND of the two bits in each entry of v; b_2 is just the most significant bit of v; b_3 can be restored by OR of the two bits in each entry of v.

Table 4.3 Bijection Between $K = 3$ Bloom Filters and 2-bit Table

$b_1(i)$	$b_2(i)$	$b_3(i)$	$v(i)$ (# of 1s)
0	0	0	0
0	0	1	1
0	1	1	2
1	1	1	3

We must point out that this k-adaptive mechanism can be used not only in Bloom filter-based IP traceback, but in other Bloom filter applications that the element number is not known a priori.

4.2.4 Theoretical Analysis and Experimental Evaluation

In this subsection, we first analyze and evaluate the traceback performance of fully deployed TOPO, and then analyze TOPO when it is partially deployed. We focus on the performance comparison between TOPO and SPIE. We finally analyze the performance of our k-adaptive mechanism for Bloom filters.

4.2.4.1 Analysis under Full Deployment of TOPO

Theoretical Analysis In the Bloom filter-based IP traceback systems, the traceback request is initiated by the victim (or IDS) when it detects intrusions. Finally the victim will get an attack graph from the IP traceback system which not only contains the real attack path, but also some extra innocent nodes because of the false positives of Bloom filters.

In SPIE's analysis [95], an upper bound of the expected number of extra nodes ex_all in the attack graph G is given by:[9]

$$E_{SPIE}(ex\_all) = \frac{Ldf}{1 - df}, \tag{4.23}$$

where d is the maximum number of each router's predecessors on the network, and L is the number of routers on the attack path. This formula requires that $0 \leq df < 1$, otherwise it will not converge. However, this formula does not answer all the questions we exactly desire

[9] We get a slight different bound in Theorem 13.

to know: How many unnecessary queries are sent out into the network? How many innocent end nodes are there in the attack graph? A good IP traceback system must have no or less unnecessary queries and innocent end nodes.

Let ex_query denote the total number of extra (unnecessary) queries that are sent out into the network, and ex_end denote the number of extra end nodes in the attack graph. We can easily get the following theorem for SPIE:

Theorem 13.

$$E_{SPIE}(ex\_query) = \frac{L(d-1)}{1-df}. \tag{4.24}$$

$$E_{SPIE}(ex\_all) = f\frac{L(d-1)}{1-df} \tag{4.25}$$

$$= f \cdot E_{SPIE}(ex\_query). \tag{4.26}$$

$$E_{SPIE}(ex\_end) = (1-f)^d f\frac{L(d-1)}{1-df} \tag{4.27}$$

$$= (1-f)^d \cdot E_{SPIE}(ex\_all). \tag{4.28}$$

Proof. Figure 4.10 demonstrates the tree structure of predecessors ($d = 3$ in this case) viewing along the reverse attack path: $Victim \rightarrow R_1 \rightarrow \ldots \rightarrow R_L \rightarrow Attacker$. In real world it should be a merged net instead of a tree, because many routers share the same predecessors. Figure 4.10 shows the worst possible scenario, since we want to calculate the upper bound of the expectations. Each level consists of the innocent nodes which have the same probability to be queried. As shown in Figure 4.10, the number of nodes on level q is

$$L(d-1)d^{q-1}. \tag{4.29}$$

According to SPIE, the probability that an innocent node on level q is queried, the probability that it is falsely included in the attack graph, and the probability that it is end node in the attack graph are

$$f^{q-1}, \tag{4.30}$$

$$f^q, \tag{4.31}$$

and

$$(1-f)^d f^q \tag{4.32}$$

respectively. Therefore, we get equation (4.24), (4.25) and (4.27). □

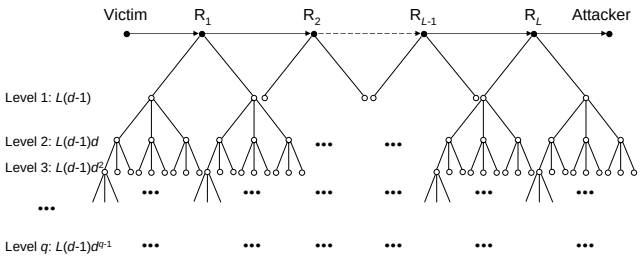


Figure 4.10 Tree Structure of Predecessors

Now we analyze the traceback performance of TOPO. Let $F = 1-(1-f)^d$. If $df << 1$, $F \approx df$. If we ignore the memory consumed by the predecessor list, we can get the following theorem.[10]

Theorem 14.

$$E_{TOPO}(ex\_query) = f \cdot E_{SPIE}(ex\_query). \tag{4.33}$$

$$E_{TOPO}(ex\_all) = F \cdot E_{SPIE}(ex\_all). \tag{4.34}$$

$$E_{TOPO}(ex\_end) = (1-fF)^d E_{TOPO}(ex\_all) \tag{4.35}$$

$$\approx F \cdot E_{SPIE}(ex\_end). \tag{4.36}$$

Proof. In TOPO, the probability that an innocent node on level q is queried, the probability that it is falsely included in the attack graph, and the probability that it is end node in the attack graph are

$$f^q, \tag{4.37}$$

[10]Please refer to the discussion in Section 4.2.5.3 to understand why the predecessor list memory size may be ignored.

$$Ff^q, \tag{4.38}$$

and

$$(1-fF)^d Ff^q \tag{4.39}$$

respectively. Therefore, we get equation (4.33), (4.34) and (4.35), and

$$\begin{aligned} E_{TOPO}(ex\_end) &= \frac{F(1-fF)^d E_{SPIE}(ex\_end)}{(1-f)^d} \\ &\approx F \cdot E_{SPIE}(ex\_end). \end{aligned} \tag{4.40}$$

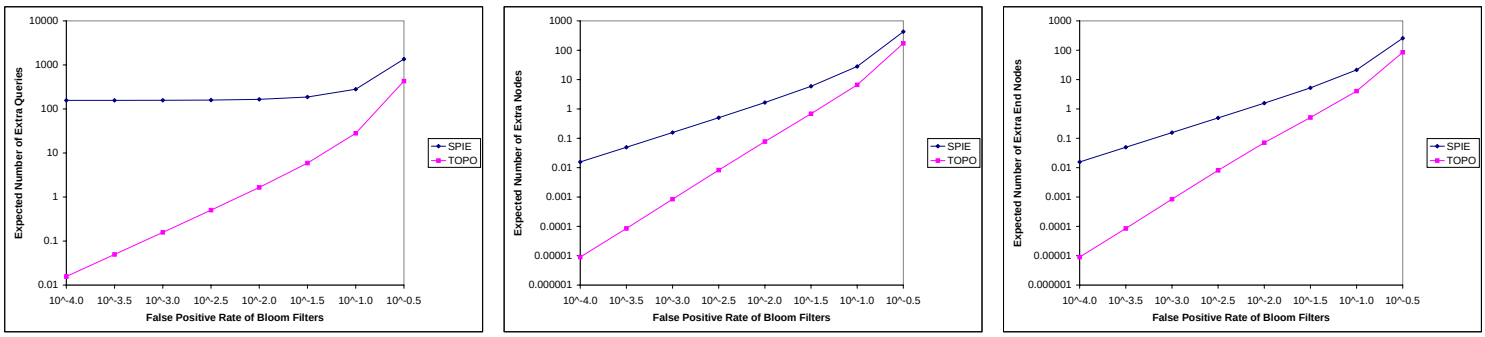
□

As indicated in Theorem 14, the number of unnecessary queries in TOPO is only f times to that in SPIE, which is a significant improvement. Also, the numbers of extra nodes and extra end nodes reduce to F times to those in SPIE. If $f = 0.0001$, and $d = 100$, $F \approx 0.01$, which means that, compared with SPIE with the same resource, only 1% nodes will appear in the attack graph using TOPO.

Experimental Study In our theoretical analysis, we have assumed that all routers have equal degree of predecessors. However, it is not realistic in real world networks, where the degrees of routers are different. In this experimental study, we use real world Internet topologies provided by CAIDA [7] to evaluate and compare the performance of SPIE and TOPO in traceback.

In our experiments, we use real world Internet topology captured on Nov. 5, 2005 from one of CAIDA's skitter monitor b-root.skitter.caida.org, which is a topology map viewed from a single origin (128.9.0.109) to 317,218 destinations[11]. Each router is assumed to be equipped with Bloom filters that have the same false positive rate. We simulate the traceback from the single origin to every destination and calculate the expected number of extra queries, extra nodes and extra end nodes with respect to the false positive rate of Bloom filters. Figure 4.11 shows our experimental results.

[11] We only consider the destinations with completed paths in this data set.



(a) Extra Queries (b) Extra Nodes (c) Extra End Nodes

Figure 4.11 Experimental Results

As shown in Figure 4.11(a), as the false positive rate f of Bloom filters decreases, both SPIE and TOPO generate less extra queries. However, when f is less than 10^{-2}, the expected number of extra queries in SPIE almost remains the same (about 156), which indicates that there exists a lower bound in SPIE on expected number of extra queries. This can be explained using equation (4.24). Meanwhile, the expected number of extra queries in TOPO always decreases as f decreases. When $f = 0.01$, the expected number of extra queries is less than 2, while SPIE's expected number of extra queries is more than 165 for the same f. Figure 4.11(b) shows the expected number of extra nodes in the attack graph. As f decreases, both SPIE and TOPO create less extra nodes. However, TOPO has much smaller expected number of extra nodes than SPIE. Figure 4.11(c) shows that TOPO also has smaller expected number of extra end nodes compared to SPIE.

In sum, both the theoretical analysis and the experimental results show that TOPO has better traceback performance compared to SPIE. In other words, TOPO can achieve the same performance as SPIE with lower memory requirement on Bloom filters. For instance, TOPO with $f = 0.005$ can achieve better traceback performance than SPIE with $f = 0.0001$. This means that TOPO requires less memory allocated for Bloom filters on routers. Therefore, TOPO is more efficient with the same traceback capability.

4.2.4.2 Analysis under Partial Deployment of TOPO

It is difficult to analyze the performance of partially deployed Bloom filter-based IP traceback systems because of the varieties of deployment. To simplify the analysis, we consider partially deployed systems with the following constraint: On all possible paths in the partially deployed system, there is at least one Bloom filter-equipped router within any S steps.

Let $E_{SPIEpd}(x)$ and $E_{TOPOpd}(x)$ denote the upper bound of the expected number of parameter x in partially deployed SPIE and TOPO respectively. When $S \geq 2$ and $d^S f < 1$, we have the following theorem:[12]

Theorem 15.

$$E_{SPIEpd}(ex\_query) = \frac{L(d^S - 1)}{1 - d^S f}. \tag{4.41}$$

$$E_{SPIEpd}(ex\_all) = \frac{L[d^{S-1} - 1 + (d-1)d^{S-1}f]}{1 - d^S f}. \tag{4.42}$$

$$E_{SPIEpd}(ex\_end) = \frac{L(d-1)d^{S-2}(1-f)^d}{1 - d^S f}. \tag{4.43}$$

$$E_{TOPOpd}(ex\_query) = E_{SPIEpd}(ex\_all). \tag{4.44}$$

$$E_{TOPOpd}(ex\_end) = \frac{L}{1-d^S f} \cdot [d^{S-2} - 1 + (d-1)d^{S-2}(df + F)] \tag{4.45}$$

$$E_{TOPOpd}(ex\_end) = \begin{cases} \frac{L(d-1)d^{S-3}(1-F)^d}{1 - d^S f}, & \text{if } S > 2 \quad (4.46a) \\ \frac{L(d-1)df(1-F)^d}{1 - d^2 f}, & \text{if } S = 2. \quad (4.46b) \end{cases}$$

We skip the proof because it is similar to the proof of Theorem 13 and 14. Theorem 15 indicates that the performance of a partially deployed SPIE or TOPO may exponentially

[12] In the analysis of partially deployed systems, we assume that there are no or little packet transformations which can be omitted on the common routers without Bloom filters.

decline as S increases. If a Bloom filter-based IP traceback system is partially deployed in a network, the Bloom filter-equipped routers should be evenly distributed among all routers.

Comparing Theorem 15 with Theorem 13 and 14, we learn that both partially deployed SPIE and TOPO would have lower performance than that of the fully deployed systems. However, partial deployment also saves the total amount of memory. If the saved memory is used to enlarge the existing Bloom filters, the false positive rate would reduce and thus help alleviate the performance losing in the partially deployed systems. Based on Theorem 15, if $d_S f << 1$, we get

Theorem 16.

$$E_{TOPOpd}(ex\_query) \approx \frac{1}{d} E_{SPIEpd}(ex\_query), \tag{4.47}$$

$$E_{TOPOpd}(ex\_all) \approx \frac{1}{d} E_{SPIEpd}(ex\_all), \tag{4.48}$$

$$E_{TOPOpd}(ex\_end) \approx$$

$$\begin{cases} \frac{1}{d} E_{SPIEpd}(ex\_end), & if\ S > 2 \quad (4.49a) \\ df E_{SPIEpd}(ex\_end), & if\ S = 2. \quad (4.49b) \end{cases}$$

Therefore, TOPO has better performance than SPIE when both of them are partially deployed in the same way. TOPO is more suitable when the IP traceback system must be partially deployed on certain networks compared with SPIE.

4.2.4.3 Analysis of k-Adaptive Mechanism

We design an experiment to analyze the performance of k-adaptive mechanism for Bloom filters. Suppose a router is designed to store traceback information within 1 hour with the granularity of 1 minute. First, it divides its memory into 61 equal slices and each slice is $1M$ bits. 1 slice is used to store the packets in the current minute, and the other 60 slices are for these 60 archived Bloom filters. The hash function number is fixed to $k = 4$. Now we divide the same memory into 63 slices and each slice is $0.968M$ bits. 3 slices are used as a table to store the values of 7 Bloom filters with different numbers of hash functions: $k1 = 1$, $k2 = 4$,

$k3 = 7$, $k4 = 10$, $k5 = 13$, $k6 = 16$, and $k7 = 19$.

Figure 4.12 shows that when packet number n varies in a large range, although the real size of single Bloom filters in our k-adaptive mechanism is smaller than the original size, our mechanism generally can achieve better performance than the original Bloom filter with fixed number of hash functions. Our performance is very close to the performance when k always equals the optimal value.

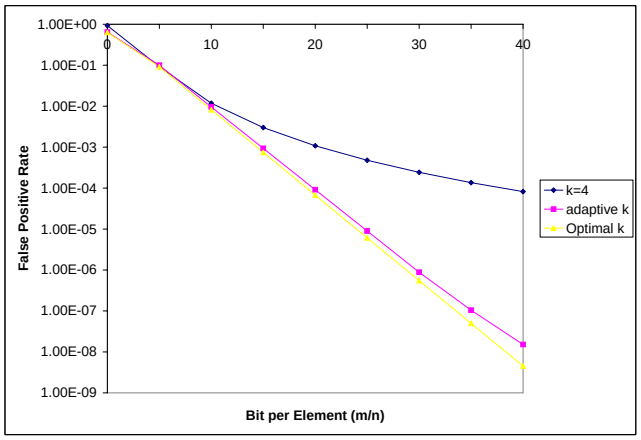


Figure 4.12 False Positive Rate Comparison

4.2.5 Further Discussions

In this section, we will discuss several considerations when designing and implementing TOPO.

4.2.5.1 Packet Signature Choice

Packet Signature can be flexible. There are different choices which can meet the requirement of distinguishing different packets. For instance, a subset of the IP header fields and the first several bytes of the packet payload are used in [74] and SPIE. PAS only uses a long excerpt of payload, which is useful when the exact packet header is unavailable. However, the excerpt must be long enough to identify different packets, and thus the attackers may avoid detect by attacking through a lot of packets with short payload. In TOPO, it is preferred to use a packet signature which contains IP header information.

4.2.5.2 Predecessor Identifier Choice

When a router has only one predecessor at each of its input port, for example, inner routers on the Internet, we choose the input port of each packet as its predecessor identifier. For a router that has multiple predecessors at one input port, we can use Layer 2 (i.e., data-link layer) information (such as source MAC addresses) to differentiate these multiple predecessors. For instance, on the Ethernet, the multiple predecessors and the router are connected through the broadcast-based transmission media. When the router receives a packet from one of its predecessors, the source MAC address of the header of the Ethernet frame can be used as the predecessor identifier. Similarly, ATM's VPI/VCI information can also be used for this purpose on ATM networks.

We propose to use local topology information in TOPO. The local topology information means the router's immediate predecessor and immediate successor. In the current system, we only use predecessor information. We believe the successor information can also be utilized. We will address this in the future research.

4.2.5.3 Predecessor List Memory Size

In most cases, the memory size of the predecessor list is much smaller than those of the Bloom filters, and that is why we ignore its influence on Bloom filters when we analyze the performance of TOPO in Section 4.2.4. Generally, inner routers do not have a large number of predecessors (which are typically no more than 100), and these predecessors often remain unchanged for a long time. Therefore, we need not archive the predecessor list often. Instead, a router may maintain a static list to store all appeared predecessor identifiers for more than one Bloom filters, and only archive 1 bit for each predecessor on the list when archiving the Bloom filters: value 1 means that the router receives packets from that predecessor in that certain time interval, and value 0 not.

To find the direct support to our point of view from the real world Internet, we analyze CAIDA's Internet Topology Data Kit #0304 (ITDK0304) [9]. ITDK0304 is the skitter data of the Internet router-level graph collected between Apr 21 and May 8, 2003. There are a total of

Table 4.4 Distribution of Internet Routers' Upstream Degrees

Upstream Degree	0 - 24	25 - 49	50 - 74	75 - 99	100 - 124	125 - 274	≥ 275	Average: 3.31
Number of Routers	190469	1501	191	52	20	11	0	Total: 192244
Percent of Routers	99.0767%	0.7808%	0.0994%	0.0270%	0.0104%	0.0057%	0	

192244 nodes and 636643 directed links. Table 4.4 shows the distribution of Internet routers' upstream degrees derived from ITDK0304. The average upstream degree is as low as 3.31, and the maximum upstream degree is only 269. Moreover, more than 99% routers have less than 25 upstream neighbors, and more than 99.98% routers have less than 100 upstream neighbors. These facts quite support our point of view that the memory sizes of predecessor lists need to be archived are adequately small compared with the large sizes of Bloom filters.

4.2.5.4 Membership Check Burden on Routers

In TOPO, suppose that there are d predecessors on one router's predecessor list. When the router receives a query message, it has to do d membership checks by combining the packet identifier with each predecessor identifier on the list, while the router in SPIE only need to do 1 membership check. It seems that TOPO complicates the membership check process, and thus aggravates the computation burden on each router. However, after using topology information in TOPO, the probability that each innocent router is queried becomes f times of that of SPIE. Let *member_check* denote the number of membership checks, and we get that

$$E_{TOPO}(member\_check) = df \cdot E_{SPIE}(member\_check). \tag{4.50}$$

In most cases, $d \cdot f << 1$, therefore actually TOPO alleviates the membership check burden on each router which receives fewer query messages and does fewer membership checks.

4.2.5.5 Applying Compressed Bloom Filters

Mitzenmacher [81] introduced *compressed Bloom filters*. He proposed that Bloom filters can be compressed to improve their performance by achieving either a lower false positive rate with the same memory size, or smaller memory size with the same false positive rate. The

compressed Bloom filters can be used to reduce the number of bits broadcast in sharing Web cache information. As shown in Table VI in [81], a compressed Bloom filter can achieve the same false positive rate as the standard Bloom filter while reducing the memory over 20%. The tradeoff costs are the increased processing requirement for compression and decompression and larger memory requirements at the endpoint machines.

In Bloom filter-based IP traceback systems, if a router stores a lot of archived pages of previous Bloom filters, and the received query messages are infrequent, the gain of applying compressed Bloom filters can overcome the processing costs introduced by compression and decompression. However, if a router only has a few archived Bloom filters, the memory overhead of implementing compression will be unacceptable. Furthermore, if the query messages are frequently received which always query different Bloom filters, the router would be busy in decompressing the required Bloom filters. The reason is that the router usually has no enough memory to keep two decompressed Bloom filters at the same time.[13]

4.2.5.6 Applying Hierarchical Bloom Filters

We also consider applying the Hierarchical Bloom filters [94] in TOPO. However, we find that actually hierarchical architecture has no benefit to false positive rate compared with the standard Bloom filters, and is even worse. The authors of [94] referred to the false positive rate of the standard Bloom filter upon which their Hierarchical Bloom filters are built as basic false positive rate f_o, and referred to the false positive rate resulting from their Hierarchical Bloom filters as effective false positive rate f_e. They showed that $f_e << f_o$. We agree with it. However, hierarchy also introduces more inserted elements into the Bloom filters, which increases the false positive rate and achieve no benefits eventually. Suppose n packets are inserted into a Hierarchical Bloom filter using q different strings for each, and the totally inserted elements are $n_0 = qn$. Then we get

$$f_e = {f_o}^q \approx (0.6185^{\frac{m}{n_0}})^q = 0.6185^{\frac{m}{n}} = f, \tag{4.51}$$

[13] Usually, the size of decompressed Bloom filters is over 10 times larger than that of the original Bloom filters.

where f is exactly the false positive rate when the standard Bloom filter inserts n packets each using just one string. Therefore there is no benefit to false positive rate using hierarchy. It makes false positive rate even worse considering that hierarchy needs to check all possible alignments of payload excerpt.

4.2.6 Conclusion

Several Bloom filter-based IP traceback schemes have been proposed. However, Bloom filters' inherent false positives restrain the effectiveness of previous schemes. In this research, we have proposed TOPO, a topology-aware single packet IP traceback system, in which the predecessor information is used for traceback purpose. Our analysis showed that TOPO significantly reduces not only the number of unnecessary queries but also the false attributions. In addition, practicability is an important and desired property of IP traceback systems. We have studied the partial deployment problem of Bloom filter-based IP traceback systems and carefully designed to allow TOPO to be partially deployed while maintaining its traceback capability. We also proposed k-adaptive mechanism for Bloom filters which control parameters may be adaptively adjusted according to the number of actual received packets. Such adjustments can help Bloom filters-based IP traceback systems to achieve the best performance in terms of false positive rate and storage space requirement when the number of arrival packets varies significantly over time.

In the future, we will continue to improve the design of TOPO in terms of processing overhead and memory space requirement.

CHAPTER 5. RESEARCH IN ONLINE FRAUD DETECTION

5.1 Introduction

With the rapid growth of the Internet, online advertisement plays a more and more important role in the advertising market. Among several online advertising models, pay-per-click model is the most popular one. However, pay-per-click model is suffering serious fraud problems: attackers earn extra incomes or deplete competitors' advertising budget by simply clicking (seldom by hands, often by automated scripts or bots) the pay-per-click advertisements without actual interest in the content of the ad's link. Such fraudulent clicks not only exhaust online advertisers' money, but also destroy the trust between online advertisers and advertising publishers, and hence damage the healthiness of online advertising market. Recently, there are several class action lawsuits against large online advertising publishers. Therefore, the development of feasible and effective solutions to click fraud problems may benefit both the advertisers and the publishers.

The source of click fraud may be from search engines, online ad publishers, ad subdistributors, competitors and web page crawlers, etc. Fraudulent clicks can be produced in various forms, such as by hands, by malicious java scripts, or by botnets. Some types of click fraud such as hit shaving problem [90] and hit inflation attacks [15], have received considerable attention recently, and several algorithms were proposed to prevent these types of click fraud.

An important issue in defending click fraud is how to deal with duplicate clicks. If we simply regard all identical clicks as fraudulent clicks, it is unfair to advertisers in some scenarios such as that an interested client visits the same ad link several times in a week. On the other hand, if the advertisers are charged for any identical clicks, then it is very easy for an attacker to make money by continuously clicking the same ad link. A reasonable tradeoff is to define a

timing threshold and only count identical clicks once within the timing window. Decaying window models, such as landmark, jumping and sliding window models, are feasible to solve this problem. However, most traditional duplicate detecting algorithms may not be directly deployed to address this problem over decaying window models.

One possible solution is to use data streaming algorithms, which have received considerable attention recently [18, 82]. Many characteristics of large data streams, such as sum, mean, variance, frequency, quantile, top-k list (hot list), distribution, etc, have been widely studied. However, the problem of duplicate detection over different decaying window models still lacks efficient and effective solutions. In this research, we will describe two effective and efficient algorithms to detect click fraud in pay-per-click streams over different kinds of decaying windows, while using as little space and operation as possible and making only one pass over the click streams.

5.1.1 Motivation

Although online advertising is an infant comparing with traditional advertising media, it grows very quickly and plays a more and more important role in the advertising market. Several studies show that more than ten billion dollars are spent in online advertising market annually [40, 60]. There are several online advertising models, such as pay-per-action, pay-per-call, pay-per-click, etc, and pay-per-click model is the most popular one among them. Online advertisers bid on keywords of search engines or ad links of online publishers such that their target links can have more chance to be visited by end users. The search engines and/or online publishers then charge advertisers based on the number of clicks. The price of a click is usually decided by the market, which varies from less than $0.01 to even above $30. However, click fraud problem heavily challenges the pay-per-click advertising model: the ad link is clicked without actual advertising impression. A survey indicates that Internet advertisers paid $0.8 billion for click fraud in 2005 and $1.3 billion estimated in 2006, and about 14.6% clicks are fraudulent [83].

The possible source of click fraud may be:

1. Search engines or online ad publishers themselves.

Since they charge advertising fees directly from advertisers, it is possible that some disreputable search engines or ad publishers generate click fraud by themselves to increase their revenue.

2. Ad sub-distributors.

The ad sub-distributors which are paid by contracted search engines or ad publishers can also benefit through generating fraudulent clicks, and they are possibly the primary source of click fraud.

3. Competitors.

The competitors may have intentions to generate click fraud to increase their rival advertisers' bill. Furthermore, since most advertisers have limited budgets on online advertising, the competitors can just click the ad links to quickly exhaust their rival companies' limited advertisement budgets such that the ad links are removed from search engines or online ad publishers sooner. For instance, if an advertiser can only afford 5000 clicks per month, then its competitors can easily generate large number of fraudulent clicks in a week to make their ad links unavailable to end clients. The competitors can even get a better position in the keyword search results with a lower price by depleting the higher bidders. There has been a reported case of such attacks [73].

4. Web page crawlers.

There are a lot of automated web page crawlers which periodically scan the Internet to update their web page databases for search, archive or other purposes. They may generate click fraud by entering the ad links inadvertently.

Fraudulent clicks can be produced by hands, by automated scripts, or by botnets, etc. Several types of click fraud, such as hit shaving problem [90] and hit inflation attacks [15], etc, have been studied, and several algorithms have been proposed to prevent such types of click fraud. However, many of them can only be deployed by the advertising publishers. Not all advertising publishers have enough motivations to deploy these algorithms, since they receive money for each click, even if it is fraudulent. Therefore, there is a conflict in detecting click

fraud: Online advertisers have enough motivations to prevent click fraud but little abilities; Online advertising publishers have more power to play a decent role in defending click fraud but without enough incentives. Such a conflict may lead to distrust between online advertisers and publishers. Recently, several class action lawsuits against large online advertising publishers appear. Google paid 90 million dollars to settle a class action lawsuit about click fraud in March 2006 [10]. In June 2006, Yahoo settled a similar lawsuit by paying 4.95 million dollars to plaintiffs' counsel, and allowing credit refund to advertisers who claim click fraud back through January 2004 [11].

A possible solution is that both the online advertisers and publishers keep on auditing the click stream and reach an agreement on the determination of valid clicks. When determining which are valid clicks in the click streams, an important issue is how to define *duplicate clicks*. Should an advertiser be charged once or twice when there are two identical clicks? Let us consider the following two scenarios.

Scenario 1: A normal client visited an advertiser's web site by clicking the ad link of a publisher. One week later, the client visited the same web site again by clicking the same ad link.

Scenario 2: The competitors or even the publishers control a botnet with thousands of computers, each of which initiate many clicks to the ad links everyday.

Obviously, the clicks in Scenario 1 should not be considered as click fraud, while those in Scenario 2 should be determined as click fraud. However, it is very difficult to identify which scenario the identical clicks belong to. A reasonable countermeasure is to prescribe that identical clicks will not count if they are within short time interval, and will count if they happen sparsely. For instance, the advertiser and the publisher can make an agreement that identical clicks will not count within one day or $100,000$ clicks. For example, suppose that it is prescribed that the same click that is 4 elements away is considered as valid click. Let $< i, t >$ denote that a click with identifier i arrives at timestamp t. Suppose we have a stream of clicks as $< i_1, 1 >, < i_2, 2 >, < i_3, 3 >, < i_1, 4 >, < i_1, 5 >, \ldots$ Then the click $< i_1, 4 >$ should be determined as a duplicate click. However, the click $< i_1, 5 >$ is a valid click although

it is identical to the previous click, since no click with identifier i_1 is counted in the previous 4 elements. Unfortunately, although detecting duplicate in a large database has been studied by many researchers, classical duplicate detecting algorithms cannot be directly utilized to address the problems in such scenarios.

Therefore, a feasible duplicate detecting algorithm should have a mechanism that is able to eliminate expired data and only consider fresh information. Decaying window models are feasible to be utilized to eliminate expired information. Although recently many algorithms are proposed for capturing different kinds of characteristics of large data streams over decaying window models [16, 17, 35, 50, 51, 65, 67, 104, 115, 117, 121], the problem of duplicate detection in data streams over decaying window models still lacks efficient and effective solutions. In the following, we will discuss a number of useful decaying window models.

5.1.2 Decaying Window Models

Decaying window models can be utilized to eliminate expired information. There are two common types of decaying windows: *count-based* windows which maintain the last (most recent) N items in the data stream, and *time-based* windows which maintain all items that arrived in the last T time units. Therefore, the time span of a count-based window may vary, while the number of items in a time-based window may change from time to time. In this research, we mainly consider count-based windows, and our algorithms can be easily extended to time-based windows. How to extend to time-based windows can be found in Section 5.2.1 and 5.3.1.

Landmark window: Landmark windows start and end once N elements arrive. When processing data streams over landmark windows, a sketch can be maintained using less memory instead of keeping all elements in current window, since all elements will expire at the same time and we can delete the expired sketch and begin a new one.

Sliding window: A sliding window, first introduced by Datar et al. [35], only contains the last N items, which is updated once a new element comes and an old element expires. Since the elements in a sliding window expire one by one, usually some timing information is

maintained to update the interested statistics when the window slides.

Jumping window: The jumping window model was first proposed by Zhu and Shasha [121]. A jumping window is a compromise between the landmark window and the sliding window. The baseline idea is to divide the entire window equally into several sub-windows, and the statistics over the entire jumping window is based on the combination of the information from the smaller sub-windows.

Generally speaking, maintaining stream information over landmark windows is easiest and requires the least memory, while maintaining streaming information over sliding windows is the most difficult and requires the most memory. However, the streaming information will have big jump when an old landmark window expires and a new landmark window begins, while a sliding window will provide more smooth information. For instance, if a click is in the end of a landmark window and an identical click happens in the beginning of the following landmark window, then this identical click will be determined as a valid click in the landmark window model.

5.1.3 Problem Statement

We first give the definition of a duplicate click in pay-per-click streams over decaying window models.

Definition 3. *A click is classified as a* duplicate click *if in the current decaying window an identical click has been determined as a valid click.*

Notice that detecting duplicates over different decaying windows on the same click stream may generate different outputs. For instance, suppose that we have a stream of clicks as $< i_1, 1 >, < i_2, 2 >, < i_3, 3 >, < i_3, 4 >, < i_3, 5 >, \ldots$ and the window size N is 3. If we apply landmark window model, then $< i_3, 4 >$ is determined as a valid click and only $< i_3, 5 >$ is reported as a duplicate. However, if sliding window model is utilized, then both $< i_3, 4 >$ and $< i_3, 5 >$ will be reported as duplicates, since $< i_3, 3 >$ has been classified as a valid click and it is in their current sliding windows.

In this research, we consider the problem stated as follows:

Given limited memory and an arbitrary window size N (N elements or N time unites), how to effectively and efficiently detect duplicates in a click stream over jumping windows or sliding windows in one pass?

We do not consider the landmark window model since many algorithms have been proposed which can be directly deployed to detect duplicate clicks over landmark windows.

5.1.4 Our Contributions

In this research, we are the first that address the problem of detecting duplicate clicks in pay-per-click streams over jumping windows and sliding windows using *significantly less* memory space and operations [116]. Since a naive deployment of classical Bloom filters to jumping windows requires many memory operations, we propose an innovative GBF algorithm using group Bloom filters which significantly reduces the memory operations when processing click streams. Our GBF algorithm is effective and efficient over jumping windows with small number of sub-windows.

However, in a jumping window when there are too many sub-windows, GBF algorithm still requires many memory operations. To solve this problem, we propose a new data structure called timing Bloom filter, which records inserted elements' timing information. We design a TBF algorithm based on timing Bloom filter which can process click streams over sliding windows and jumping windows using less memory space and processing time.

One advantage of our GBF algorithm and TBF algorithm is that both of them have no false negative. Furthermore, both theoretical analysis and experimental results show that our algorithms are effective and efficient which can achieve low or bounded false positive rate when detecting duplicate clicks in pay-per-click streams over jumping windows and sliding windows.

5.2 Detecting Duplicates over Jumping Windows Using Group Bloom Filters

5.2.1 GBF Algorithm Description

To detect duplicates in click streams over a landmark window, Bloom filters can be directly deployed [75]. Each click has an predefined identifier, such as the source IP address, or the cookie, etc. Then each click's identifier is hashed into the Bloom filter. If a click's identifier is present in the Bloom filter before insertion (i.e., all corresponding bits are 1s), then it is reported as a duplicate.

To detect duplicates in pay-per-click streams over jumping windows, a naive solution is to evenly divide the entire jumping window into a number of sub-windows and maintain a separate Bloom filter for each sub-window. To save operation time, all Bloom filters should use the same set of hash functions. Suppose N is the size of the jumping window which is divided into Q sub-windows. After the jumping window is full, there will be an expired Bloom filter after each $\frac{N}{Q}$ elements. Therefore, if we want to use the memory space of the expired Bloom filter for the upcoming sub-window, we must clean the entire expired Bloom filter before the first element of the upcoming sub-window can be inserted. However, considering that cleaning an expired Bloom filter need $O(m)$ operations, where m is the size of the Bloom filters, we must keep the newly arrived elements in an extra queue before we finish the clean operations. To solve this problem, we can divide the total available memory space into $Q+1$ pieces. Q pieces are for the Bloom filters of Q active sub-windows, and the additional piece is used to maintain a Bloom filter for the elements in the upcoming sub-window while we clean the expired Bloom filter. Then we have more time to clean the expired Bloom filter, since we only need to clean the expired memory before the next Bloom filter expires. Let M denote the total number of memory bits, then each Bloom filter has size $m = \frac{M}{Q+1}$, and we only need to clean $\frac{M/(Q+1)}{N/Q} = \frac{QM}{(Q+1)N}$ bits when processing each newly arrived element.

When a new element comes, it is inserted into the Bloom filter of the corresponding sub-window if and only if it is not a duplicate in the current jumping window (including all active

sub-windows). To check whether it is a duplicate, we first calculate k hash values using the element's identifier. We then have to check each of the Q active Bloom filters (suppose the jumping window is full) by reading the corresponding k bits. If the set of k bits in any Bloom filter is all 1s, then this element is reported as a duplicate click; otherwise, the corresponding k bits in the Bloom filter of the current sub-window are set to 1.

Obviously, such a duplicate-checking procedure may cost about $(Q \times k)$ memory operations, which is very time consuming if Q is large. To solve this problem, we introduce a data structure called *Group Bloom Filters* (GBF) which can significantly reduce required memory operations. The baseline idea is that instead of dividing the entire memory into separate pieces for separate Bloom filters, the bits with the same index in each Bloom filter are grouped together in GBF. Then using this data structure, the CPU can visit the required bits in a bunch. For instance, suppose that $Q = 31$ and the size of a word in the memory is 32 bits. Then the same bits of the total 32 Bloom filters will be in the same word in the memory. Suppose that CPU can read/write one 32-bit word each time, then we can fetch all bits we need using k memory reads. After we get the k 32-bit words, we AND them into a single 32-bit word. We then mask the bit which represents the expired Bloom filter in the word by setting the corresponding bit to 0. If the value of this word is none-zero, then the new element is a duplicate; otherwise, we set each corresponding bit for current sub-window in the k 32-bit words to 1 and write them back to the memory.

Figure 5.1 shows the description of our GBF algorithm to detect duplicates over jumping windows using group Bloom filters. Each bit in GBF is initialized to 0 before processing the date stream, and $W[i]$ denotes the word with index i in GBF. Concurrently with the finish of the new sub-window after processing $\frac{N}{Q}$ elements, the expired Bloom filter is cleaned and ready to insert new elements. A counter can be used to determine when a sub-window is full and a new sub-window starts.

As an example, Figure 5.2 shows how GBF algorithm works. Let $x_0, x_1, x_2, \cdots$ denote a series of clicks. Suppose that the jumping window has size $N = 6$, which is divided into $Q = 3$ sub-windows, and thus each sub-window contains 2 clicks. We maintain 4 Bloom filters with

Step 1: Clean Expired Bloom Filter.

Starting after last cleaned word, set the corresponding bits of the expired Bloom filter of next $\frac{QM}{(Q+1)N}$ words to 0.

Step 2: Process New Element x_t.

Set temporary word W' to all 1s except that the bit which represents the expired Bloom filter is set to 0.

for $i \leftarrow 1$ to k
  $W' \leftarrow W'$ AND $W[h_i(x_t)]$
endfor
if $W' \neq 0$
  x_t is a duplicate click.
else
  for $i \leftarrow 1$ to k
    Set the corresponding bit of the current sub-window in $W[h_i(x_t)]$ to 1.
  endfor
endif

Figure 5.1 GBF Algorithm Description

size $m = 8$, and the hash functions are $h_1()$ and $h_2()$. Click x_0 and x_1 will be inserted into the Column C1, and x_2 and x_3 will be inserted into the Column C2, and so on. Figure 5.2(a) shows the status when the $11th$ click x_{10} is coming. Then x_{10} will be inserted into Column C2. In the beginning, we will execute Step 1 to clean the first 4 bits ($[b0 : b3]$) of expired Column C3. Then we hash x_{10} using $h_1()$ and $h_2()$ and get indices 7 and 4. We AND words $W[7]$ and $W[4]$ together and get word $W' = [0100]$. We then mask the corresponding bit of Column C3 which is expired, and get $W' = [0000]$. Therefore, x_{10} is a valid click and bit b7 and b4 in Column C2 are set to 1. The gray bits shown in Figure 5.2 indicate that these bits are changed after processing a new element. When the $12th$ click x_{11} is coming, we first clean the last 4 bits ($[b4 : b7]$) of Column C3. We then calculate and get two indices 1 and 5. This time $W' = [1000]$. Therefore, x_{11} is reported as a duplicate, and not inserted into Column C2. When the $13th$ click x_{12} is coming, notice that now Column C4 is expired and Column C3 is entirely cleaned and ready to insert new elements.

Extension to Handle Time-Based Windows

GBF algorithm can be easily extended to handle time-based jumping windows. Instead of

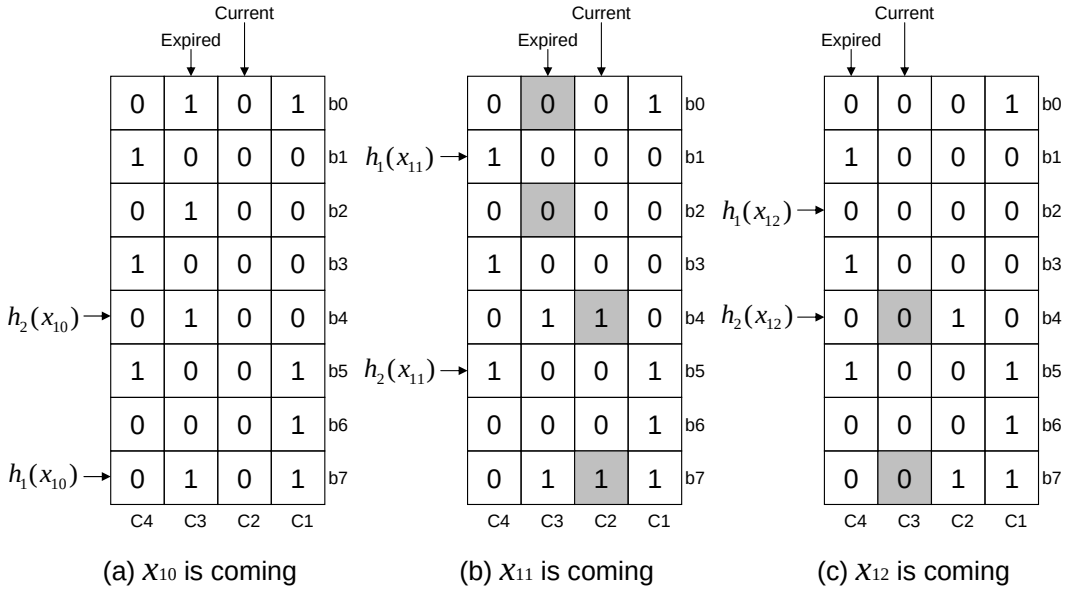


Figure 5.2 An Example of GBF Algorithm

dividing entire jumping window equally by counting elements, the time-based jumping window is divided into Q sub-windows with same time expansion. Then each sub-window is equally divided into R time units. In Step 1, the cleaning procedure executes once in each time unit, and scans $\frac{M}{(Q+1)R}$ entries. Step 2 stills executes once a new element comes.

5.2.2 Theoretical Analysis

When designing algorithms to detect duplicate clicks in pay-per-click streams over jumping windows, we must consider the false negative rate and false positive rate, the memory requirement, and the processing time. The following theorem provides the properties of our GBF algorithm.

Theorem 17. *Let N denote the size of the jumping window, which is divided into Q sub-windows. Given M-bit memory, and assuming that the CPU can read/write a D-bit word in each cycle, group Bloom filters can detect duplicates over jumping windows with the following properties:*

1. *The false negative rate is 0.*
2. *The false positive rate is $O(Q \cdot 0.6185^{\frac{M}{N}})$.*

3. *The running time to process each element is* $O(\frac{Q}{D} \cdot \frac{M}{N})$ *in worst case.*

Proof We proof the performance of GBF algorithm in terms of false negative rate, false positive rate and running time as follows:

False Negative Rate

False negatives mean that the detection of duplicate clicks is falsely missed. According to the definition of duplicate over decaying window models, a click is a duplicate if in current decaying window an identical click has already been determined as a valid click. Then there are two possible cases when checking a duplicate click x_t:

Case 1: A previous identical click is accurately classified as valid click and has been inserted into GBF.

According to GBF algorithm, since all k corresponding bits have been set to 1, then the click x_t will be reported as duplicate accurately.

Case 2: A previous identical click is falsely classified as duplicate and is not inserted into GBF due to the inherent false positive property of Bloom filters.

In this case, it is possible that GBF algorithm determines click x_t as valid. However, since the previous valid identical click has been reported as a duplicate, such a missing detection actually has no effect on the overall result.

False Positive Rate

To simplify the analysis, we assume that all Bloom filters of the sub-windows have the same false positive rate f_0. Since the entire jumping window with N elements is evenly divided into Q sub-windows, then each sub-window contains $n = \frac{N}{Q}$ elements. GBF algorithm divides the total memory into $Q + 1$ pieces, then the size of the Bloom filters of the sub-windows is $m = \frac{M}{Q+1}$. Since GBF algorithm inserts an element if and only if an identical element is not present in the current window, then actually each Bloom filter in GBF has less elements inserted compared with a classical Bloom filter. Consequently, each Bloom filter in GBF is similar to a classical Bloom filter with size m into which N elements are inserted, but has better false positive rate. To simplify the analysis, we just assume that it has the same false positive rate as a classical Bloom filter. According to equation (5.1), if all elements in a sub-window

are distinct and inserted into the Bloom filter, and the number of hash functions, k, is set to the optimal value, we get

$$f_0 \approx 0.6185^{\frac{m}{n}} = 0.6185^{\frac{QM}{(Q+1)N}}. \tag{5.1}$$

In GBF algorithm of detecting duplicates over jumping windows using GBF, an innocent element is falsely reported as a duplicate if and only if it is present in any of the Q active Bloom filters. Suppose that all Q sub-windows are active, and let f' denote the false positive rate of the most recent Bloom filter which may be not full. Then we have

$$\begin{aligned} f_{GBF} &\approx 1-(1-f_0)^{Q-1}\cdot f' \approx 1-(1-f_0)^{Q} && (5.2)\\ &\approx 1-(1-0.6185^{\frac{QM}{(Q+1)N}})^{Q} && (5.3)\\ &\approx 1-(1-0.6185^{\frac{M}{N}})^{Q} && (5.4) \end{aligned}$$

When $0.6185^{\frac{M}{N}}$ is small, we have

$$f_{GBF} \approx Q\cdot 0.6185^{\frac{M}{N}}. \tag{5.5}$$

Therefore, given M-bit memory and jumping window size N which is divided into Q sub-windows, the false positive rate of our GBF algorithm is $O(Q\cdot 0.6185^{\frac{M}{N}})$.

Running Time

As shown in the description of GBF algorithm, when a new element arrives, we only read/write $\frac{2QM}{(Q+1)N}$ words or $\frac{2QM}{N}$ bits in GBF to eliminate expired information in Step 1. Since each time the CPU can read/write D bits, then there are $O(\frac{Q}{D}\cdot\frac{M}{N})$ read/write operations. Furthermore, to insert a new element, we only check k entries which indices are calculated by k hash functions. Suppose that the running time to calculate a hash value is $O(1)$, then the running time of Step 2 is $O(k)=O(\frac{M}{N})$. Therefore, the running time to process each element is $O(\frac{Q}{D}\cdot\frac{M}{N})$ in worst case.

5.2.3 Comparison with Previous Work

In [75], the authors proposed to maintain a counting Bloom filter for each sub-window, and a main Bloom filter which is a combination of all counting Bloom filters and represents the entire jumping window. When a new sub-window is generated, the eldest window is expired and subtracted from the main Bloom filter. Combining two counting Bloom filters is performed by adding the corresponding counters; deleting an old counting Bloom filter is performed by subtracting its counters from the main Bloom filter.

However, this scheme has two potential drawbacks. One is that subtracting an expired Bloom filter from the main Bloom filter needs $O(m)$ operations, and false positives increase if new elements are inserted into the main Bloom filter before subtracting operation completes. The other drawback is that this scheme may have high false positive rate, especially when the number of sub-windows is large. There are two reasons for this drawback. First, with the same limited available memory space, expanding bits in Bloom filters to counters make the size of Bloom filters smaller. In worst case, the maximum value in the counters of counting Bloom filters is $\frac{N}{Q}$, and the maximum value in the counters of the main Bloom filter is N. Therefore, each counter must have enough bits to avoid saturation, which will generate both false negatives and false positives. Consequently, the size of the Bloom filters in their algorithm is much smaller than the size of Bloom filters in our GBF algorithm. According to equation (5.1), the false positive rate will be much higher than that of GBF algorithm. Second, checking the presence of an element in the main Bloom filter which is the result of combination of all counting Bloom filters will generate very high false positive rate, since it is as if all N elements are inserted into the single main Bloom filter (any entry with a non-zero value in any counting Bloom filter will set the corresponding entry in the main Bloom filter to non-zero). On the contrary, each Bloom filter in GBF algorithm is only inserted at most $\frac{N}{Q}$ elements.

Figure 5.3 shows the comparison between the algorithm in [75] and our GBF algorithm. We draw the false positive rate when $Q = 31$, $m = 2^{20}$, and N increases from 2^{15} to 2^{21}. We make the observation that with the increasing window size, the false positive rate of the algorithm in [75] increases more quickly compared with GBF algorithm when both algorithms

maintain Bloom filters with the same size. For instance, when $N = 2^{20}$, the false positive rate of the algorithm in [75] is about 0.62, while the false positive rate of GBF algorithm is only about 0.000011. Notice that when both algorithms maintain Bloom filters with the same size, the algorithm in [75] requires more memory space since its Bloom filters are counting Bloom filters which contain more bits in each entry.

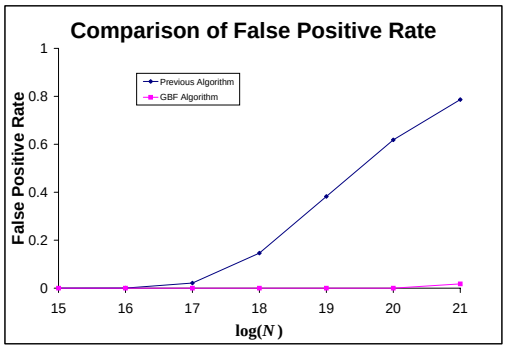


Figure 5.3 Comparison Between Previous Algorithm and GBF Algorithm

5.3 Detecting Duplicates over Sliding Windows Using Timing Bloom Filters

GBF algorithm works well over jumping windows with small number of sub-windows. However, there is a limitation of GBF algorithm that it is not feasible in the sliding window model, or when the number of sub-windows Q is very large in a jumping window. In sliding windows, although we can keep N Bloom filters, each of which only hold one element, maintaining N Bloom filters make the running time unacceptable. For instance, suppose the window size $N = 2^{20}$, and the CPU can read/write 64 bits per cycle. Before an element is inserted into the Bloom filter for that sub-window, $16384 \cdot k$ reads must be executed, where k denotes the number of hash functions. Therefore, the GBF algorithm cannot process high-speed click streams over sliding windows or jumping windows with large number of sub-windows. We hence devise an innovative TBF algorithm based on a new data structure called *Timing Bloom Filters* (TBF) to detect duplicate clicks over sliding windows and jumping windows with large number of sub-windows.

5.3.1 TBF Algorithm Description

We propose a new data structure called timing Bloom filters which contain timing information derived from classical Bloom filters. The timing information contained in TBF can be utilized to evict stale data out of our data structure, and make TBF applicable to process data streams over sliding windows. Let N denote the sliding window size. An existing element is called *active* if it is one of the most recent N elements within the current window, or *expired* if it left the current window. For each element x_i, an index pos_i is used to record its timestamp (i.e., position) in the data stream, which is an indicator of "active" or "expired" by comparing with pos – the position index of the most recent element.

Our new data structure TBF is based on Bloom filters. To insert timing information into TBF, each bit in the classical Bloom filter is replaced by an entry with $O(\log N)$ bits. At the beginning of the click stream, all bits in all entries of the TBF are initialized to bit 1. When a new element arrives, we first calculate the k hash functions and get (at most) k indices. We then check the corresponding k entries to judge if this element is both present[1] in the TBF and active[2] in the current sliding window. If it is present and active, then we just ignore it and report it as a duplicate click; otherwise, we set the corresponding k entries using this element's timestamp. The timestamps are represented by wraparound counters, and the number of bits in each entry of TBF is set to be large enough such that no timestamp is represented by all 1s.

Our TBF algorithm has two steps when a new element arrives. Step 1 deletes expired information in TBF; Step 2 processes the new element. When a new element x_t comes, the current timestamp pos is updated. Usually there is an old element expired (except at the beginning of the click stream when the sliding window is not full). Therefore, besides processing the new arriving element, the expired timestamps in the TBF must be removed, which means TBF algorithm only maintains the timestamps of active elements in the current window. To bound the bits to represent the timestamp in the continuous click stream, we have to use a wraparound counter. We first consider the scenario that we use a wraparound counter with maximum $N-1$ to represent the timestamps, that is, the N-th element's timestamp is 0 instead

[1] "Present" means no entry in the k corresponding entries is all 1s.
[2] "Active" means all timestamps in these k corresponding entries are within the current sliding window.

of N. Suppose the newly arrived element's timestamp is P, then in this scenario the expired element also has timestamp P. Therefore, before inserting the new element into the TBF, we must first remove these expired timestamps if they have been inserted before. However, since the expired element is not maintained in memory, we have no knowledge about where the potential k expired timestamps are in the TBF. Consequently, we must scan the entire TBF with m entries to find these expired timestamps and replace them by all 1s. Since this is time consuming which needs $O(m)$ operations, such an algorithm cannot process high-speed click streams. We therefore propose an advanced update mechanism which only uses $O(\frac{m}{N})$ operations and only consumes very small additional space.

In our update mechanism, instead of setting $N-1$ as the maximum value in the wraparound counter, we set $N+C-2$ as the maximum timestamp, where C is a positive integer. Since now we expand the range of timestamp representation, we get extra time to remove stale timestamps and thus less entries in TBF need to be scanned each time. As discussed above, if $C=1$, we have to scan entire m entries when processing each new element. If $C=2$, we only need to scan half of the m entries. Therefore, when a new element arrives, we only need to check $\frac{m}{C}$ entries in TBF. The entire TBF will be scanned thoroughly after C elements arrive.

The choice of value of C is flexible. Since $\lceil \log(N+C) \rceil$ is the number of bits in an entry to represent timestamps, and $\frac{m}{C}$ is the number of entries that need to be scanned per element to update the TBF, then a smaller C means less space requirement and larger operation time, and a larger C means larger space requirement and less operation time. In the following analysis and experiments, we typically choose C equal to N.

Figure 5.4 shows the description of TBF algorithm to detect duplicates over sliding windows using timing Bloom filters. In the algorithm description, i_{pre} is initialized to 0 before processing the click stream, which records the index of entry that has been scanned previously; $P[i]$ denotes the timing information (i.e. position or timestamp) in entry i of TBF; $flag_{dup}$ indicates whether the newly arrived element is a duplicate or not.

As an example, Figure 5.5 shows how TBF algorithm works. Let $x_0, x_1, x_2, x_3, x_4, x_5, \cdots$ denote a series of clicks. Suppose that the sliding window has size $N=4$. We maintain a TBF

Step 1: Delete Expired Information.

$pos \leftarrow (pos + 1) \bmod (N+C-1)$
for $i \leftarrow i_{pre}$ to $i_{pre} + \frac{m}{C}$
  if $(pos - P[i \bmod m]) \bmod (N+C-1) \geq N$
    $P[i \bmod m] \leftarrow [11...1]$
  endif
endfor
$i_{pre} \leftarrow (i_{pre} + \frac{m}{C}) \bmod m$

Step 2: Process New Element x_t.

$flag_{dup} \leftarrow 1$
for $i \leftarrow 1$ to k
  if $P[h_i(x_t)] = [11...1]$
      or $(pos - P[h_i(x_t)]) \bmod (N+C-1) \geq N$
    $flag_{dup} \leftarrow 0$
    break
  endif
endfor
if $flag_{dup} = 1$
  x_t is a duplicate click.
else
  for $i \leftarrow 1$ to k
    $P[h_i(x_t)] \leftarrow pos$
  endfor
endif

Figure 5.4 TBF Algorithm Description

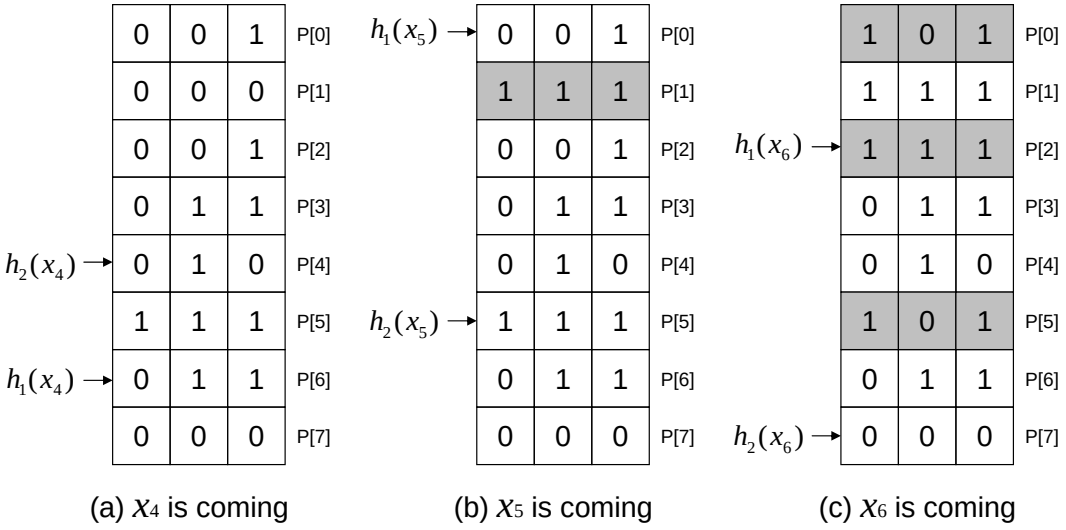


Figure 5.5 An Example of TBF Algorithm

with size $m = 8$, and each entry has 3 bits. The hash functions are $h_1()$ and $h_2()$. We set $C = 4$, which means we only need to scan and update 2 entries when processing each element. Figure 5.5(a) shows the status when the $5th$ click x_4 is coming. Notice that all entries with position [000] are expired. In the beginning, we execute Step 1 to scan the first 2 entries, and $P[1]$ is set to all 1s. Then we hash x_4 using $h_1()$ and $h_2()$ and get indices 6 and 4. Since both $P[6]$ and $P[4]$ are present and active, x_4 is reported as a duplicate. The gray entries shown in Figure 5.5 indicate that these entries are changed after processing a new element. When the $6th$ click x_5 is coming, we first scan entries $P[2]$ and $P[3]$, and this time $P[2]$ is set to all 1s. We then calculate and get two indices 0 and 5. Since $P[0] = [001]$ is expired and $P[5]$ is all 1s, x_5 is determined as a valid click, and its position information is inserted into $P[0]$ and $P[5]$. Notice that although $P[7] = [000]$ has been expired after x_4 comes, it may remain in the memory until x_7 comes. However, its presence does not affect duplicate detection in TBF algorithm.

Extension to Handle Time-Based Windows

TBF algorithm can be easily extended to handle time-based sliding windows. Suppose the entire sliding window is equally divided into R time units. In Step 1, the cleaning procedure executes once in each time unit, and scans $\frac{m}{R}$ entries. Step 2 stills executes once a newly

arrived element comes. However, instead of inserting the counting-based position, the time unit information is inserted into the entries of TBF.

Furthermore, TBF can also be easily extended to handle jumping windows. If TBF is utilized over a jumping window which is evenly divided into Q sub-windows, then all elements in the same sub-window will have the same timestamp, and they will be eliminated from TBF simultaneously. When Q is large, GBF cannot process the click stream efficiently, and TBF is a better choice.

5.3.2 Theoretical Analysis

The following theorem provides the properties of our TBF algorithm when detecting duplicate clicks in pay-per-click streams over sliding windows.

Theorem 18. *Let N denote the size of the sliding window. Given M-bit memory, timing Bloom filters can detect duplicates over sliding windows with the following properties:*

1. *The false negative rate is 0.*
2. *The false positive rate is $O(0.6185^{\frac{M}{N \log N}})$.*
3. *The running time to process each element is $O(\frac{M}{N \log N})$ in worst case.*

proof We proof the performance of TBF algorithm in terms of false negative rate, false positive rate and running time as follows:

False Negative Rate

Since the proof is similar to that in Theorem 17, we skip this part to save space.

False Positive Rate

As shown in TBF algorithm description, TBF only keeps the timestamps of the most recent N elements, and all expired timestamps will be removed in time. Therefore, in worst case there are N distinct elements' timestamps present in TBF when we process a newly arrived element.[3] Since TBF algorithm inserts an element if and only if an identical element is not present in the current window, then actually TBF has less elements inserted compared with a classical

[3] Although some expired timestamps can survive in TBF for a certain time before they are cleaned, they do not affect the result when determining whether a new element is a duplicate or not.

Bloom filter. Consequently, TBF is similar to a classical Bloom filter with size m into which N elements are inserted, but should have better false positive rate. To simplify the analysis, we just assume that TBF has the same false positive rate as a classical Bloom filter. Suppose that we set $C = N$, then each entry occupies $\lceil \log 2N \rceil$ bits, and $m = \frac{M}{\lceil \log 2N \rceil}$. Similar to equation (5.1), when the number of hash functions, k, is set to the optimal value, the false positive rate of TBF is

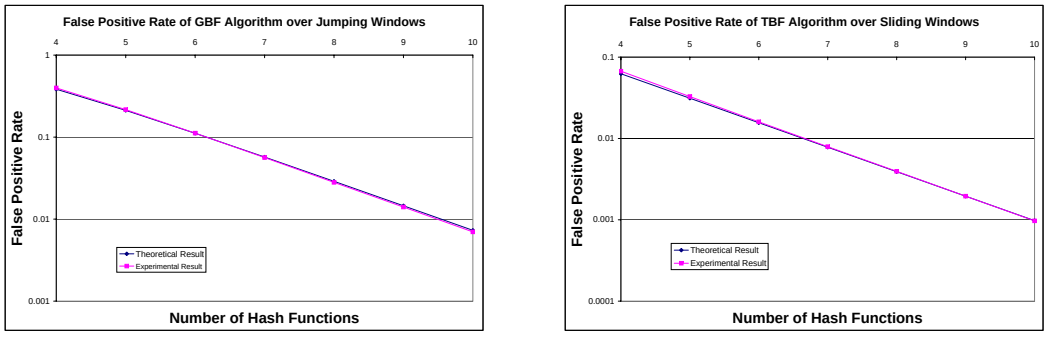
$$f_{TBF} \approx 0.6185^{\frac{m}{N}} \approx 0.6185^{\frac{M}{N \log 2N}}. \tag{5.6}$$

Therefore, the false positive rate of TBF is $O(0.6185^{\frac{M}{N \log N}})$.

Running Time

As shown in our TBF algorithm description, each time when a new element arrives, we only remove expired timestamps in TBF by scanning $\frac{m}{C}$ entries in Step 1. Generally C is $O(N)$, and $O(\frac{m}{C}) = O(\frac{M}{N \log N})$. Furthermore, to insert a new element, we only check k entries which indices are calculated by k hash functions. Assuming that the running time to calculate a hash value is $O(1)$, then the running time of Step 2 is $O(k) = O(\frac{M}{N \log N})$. Therefore, the running time to process each element is $O(\frac{M}{N \log N})$ in worst case.

5.4 Experimental Evaluation



(a) False Positive Rate of GBF (b) False Positive Rate of TBF

Figure 5.6 False Positive Rate of GBF and TBF Algorithm over Sliding Windows

In this section, we evaluate GBF algorithm and TBF algorithm for duplicate click detection over jumping windows and sliding windows. Since our algorithms have no false negative, we

only ran experiments to evaluate the false positive rate of our algorithms. Hence, we simulate our algorithms by processing synthetic click streams which have no duplicate click.

We first consider the theoretical results. Our algorithms are a little different comparing with the classical Bloom filters. In our algorithms, we only insert an element if and only if an identical element is not present in the current window. Therefore, let f denote the false positive rate of our algorithm, and then only about $(1-f)N$ expected elements are inserted. In our GBF algorithm for jumping windows, let f_0 denote the false positive rate of the sub-windows, and f_{GBF} denote the overall false positive rate. Then

$$\begin{aligned} f_{GBF} &\approx 1-(1-f_0)^Q && (5.7)\\ &\approx 1-(1-(1-e^{-(1-f_{GBF})kn/m})^k)^Q && (5.8)\\ &\approx 1-(1-(1-e^{-(1-f_{GBF})\frac{kN}{Qm}})^k)^Q && (5.9) \end{aligned}$$

For given k, if we set

$$m=\frac{(1-2^{-k})^Q kN}{Q\ln 2}, \tag{5.10}$$

then

$$f_{GBF}\approx 1-(1-2^{-k})^Q. \tag{5.11}$$

In our TBF algorithm for sliding windows, let f_{TBF} denote the overall false positive rate. Then

$$f_{TBF}\approx (1-e^{-(1-f_{TBF})kN/m})^k. \tag{5.12}$$

For given k, if we set

$$m=\frac{(1-2^{-k})kN}{\ln 2}, \tag{5.13}$$

then

$$f_{TBF}\approx 2^{-k}. \tag{5.14}$$

In the experiments of evaluating our GBF algorithm over jumping windows, we set the jumping window size to $N=2^{20}$, and the number of sub-windows to $Q=8$. For given k,

the size m of each Bloom filter is set using equation (5.10). We generated $20 \cdot N$ distinct click identifiers. We counted the false positives within the last $10 \cdot N$ clicks to make sure that GBF has been stable. Figure 5.6(a) shows the theoretical and experimental results. We make the observation that the experimental result of GBF algorithm is very close to the theoretical result when detecting duplicates over jumping windows. When $k = 10$ and $m = 1,876,246$, the false positive rate is only about 0.007.

In the experiments of evaluating our TBF algorithm over sliding windows, we set the sliding window size to $N = 2^{20}$. For given k, the size m of TBF is set using equation (5.13). We generated $20 \cdot N$ distinct click identifiers, and counted the false positives within the last $10 \cdot N$ clicks to make sure that the TBF has been stable. Figure 5.6(b) shows the theoretical and experimental results. We make the observation that the experimental result of TBF algorithm is very close to the theoretical result when detecting duplicate clicks over sliding windows. When $k = 10$ and $m = 15,112,980$, the false positive rate is only about 0.001.

5.5 Conclusions

In this research, we address the problem of detecting duplicate clicks in pay-per-click streams over jumping windows and sliding windows. We propose group Bloom filters which significantly reduces the memory operations when processing click streams, and our GBF algorithm based on group Bloom filters is effective and efficient over jumping windows. To detect duplicate clicks over sliding windows and jumping windows with large number of sub-windows, we propose an innovative TBF algorithm based on a new data structure called timing Bloom filter, which can process click streams over sliding windows using less memory space and processing time. Both theoretical analysis and experimental results show that our algorithms are effective and efficient in terms of false negative rate, false positive rate and running time when detecting duplicate clicks in pay-per-click streams.

In the future, we will continue to explore the issues of click fraud and click quality under data stream models. We will consider various sophisticated click fraud attacks, and study advertising network dynamics, new service models, economic and social impacts of click frauds.

CHAPTER 6. SUMMARY

6.1 Conclusion

In this dissertation, we design effective techniques for detecting and attributing cyber criminals. We consider two kinds of fundamental techniques: forensics-sound attack monitoring and traceback, and forensics-sound online fraud detection. Our proposed techniques may serve as fundamental components which can be widely utilized not only in network security, but also in many other domains, such as database, data mining, computer graphics, etc. The contributions of our research are as follows:

(1) We propose several innovative algorithms which answer some open problems in fundamental statistics estimation over sliding windows. Those algorithms can be used to detect anomaly and attacks in networks. We also propose efficient and effective algorithms which can trace back stepping stone attacks and single packet attacks.

- We study the problem of maintaining ϵ-approximate variance of data streams over sliding windows. We propose the first ϵ-approximation algorithm that is optimal in both space and worst case time. Our algorithm requires $O(\frac{1}{\epsilon}\log N)$ space and $O(1)$ running time in worst case.

- We address the problem of estimating ϵ-approximate frequency in data streams over sliding windows. We propose the first efficient deterministic algorithm which can achieve $O(\frac{1}{\epsilon})$ space requirement and only need $O(1)$ running time to process each item in the data stream and to answer a query.

- we consider the problem of estimating ϵ-approximate diameter, convex hull and skyline in data streams over sliding windows. We first present an improved algorithm which

only requires $O((\frac{1}{\epsilon})^{\frac{d+1}{2}} \log R)$ space to estimate the diameter over sliding windows. We then extend our algorithm to solve convex hull estimation problem. Finally, we propose a novel algorithm to estimate skyline which requires $O(\frac{1}{\epsilon^d} \log \epsilon R)$ space.

- Several algorithms are proposed to attribute stepping stone attackers. Our schemes can effectively detect stepping stones even when delay and chaff perturbations exist simultaneously.

- A topology-aware single packet IP traceback system, namely *TOPO*, is proposed to traceback single packet attacks. We design TOPO to allow partial deployment while maintaining its traceback capability. A k-adaptive mechanism is designed which can dynamically adjust parameters of Bloom filters to reduce the false positive rate.

(2) We propose streaming algorithms to detect click fraud in pay-per-click streams of online advertising networks.

- We address the problem of detecting duplicate clicks in pay-per-click streams over jumping windows and sliding windows, and propose two innovative algorithms that make only one pass over click streams and require *significantly less* memory space and operations. A patent [55] is pending based on our research.

6.2 Future Work

6.2.1 Data Stream Processing

Although our research in this dissertation have answered several open problems in data stream processing, there still have many unsolved problems in this area.

Currently, basic-counting and variance estimation over sliding windows have been researched and optimal algorithms were proposed. However, higher-ordered moments estimation problems still lack efficient algorithms.

In geometric computation area, although many algorithms were presented, we still don't know the lower bound of high-dimensional diameter, convex hull and skyline, and thus we don't

know whether current algorithms are optimal in space requirement or not. Further research is required in this topic.

6.2.2 Attack Traceback

There are still some open problems in attack detection and traceback.

6.2.2.1 VoIP Attribution

Like the Internet, the VoIP also provides unauthorized services. Therefore, some security issues existing in the Internet may also appear in the VoIP systems. For instance, a phone user may receive a call with a qualified caller ID from his/her credit card company, so he/she would answer the critical questions about social security number and date of birth, and so on. However, this call comes actually from an attacker who fakes the caller ID using a computer. Compared with a PSTN phone or mobile phone, IP phone lacks monitoring. Therefore, it is desirable to provide schemes that can attribute or trace back to the VoIP callers.

6.2.2.2 Botnet Traceback

A botnet is a network of compromised computers, or bots, commandeered by an adversarial botmaster. Botnets usually spread with virus and communicate through Inter Relay Channel (IRC) [29]. With the army of bots, the bot controllers can launch many attacks, such as spam, phishing, key logging, and denial of service. Now, more and more scientists are interested in how to detect, mitigate, and trace back botnet attacks.

6.2.2.3 Traceback in Anonymous Systems

Another issue is that a lot of anonymous systems, such as Tor [8], exist all over the world. Tor is a toolset for anonymizing web browsing and publishing, instant messaging, IRC, SSH, and other applications that use the TCP protocol. It provides anonymity and privacy for legal users, and at the same time, it is a good platform to launch stepping stone attacks. Communications over Tor are relayed through several distributed servers called onion routers.

There are more than 800 onion routers all over the world so far. Since Tor may be seemed as a special stepping stone attack platform, it is interesting to consider how to trace back attacks over Tor.

6.2.3 Online Fraud Detection

Online auction networks, such as eBay, have attracted a lot of buyers and sellers. More and more people begin to purchase goods and services through eBay or other online auction networks. However, fraud transactions happen everyday. In these online auction systems, users are often intent on doing transactions with accounts who have high feedbacks. Nevertheless, fraudsters can easily bypass the feedback systems. One method for a fraudster to build excellent feedback is to construct a clique and prompt each other. The cliques of accomplices can easily prompt many fraudsters, while never jump out to cheat directly by themselves, which makes it difficult to detect these cliques.

Two preliminary papers [87, 86] have been finished with Yanlin Peng[1] and Dr. Yong Guan, and further research is ongoing.

[1]Yanlin Peng did the majority work including algorithm design, implementation and writing.

BIBLIOGRAPHY

[1] Internet World Stats. [Online]. Available: http://www.internetworldstats.com

[2] NLANR Trace Archive. [Online]. Available: http://pma.nlanr.net/Traces/long/

[3] LAND Attack. [Online]. Available: http://www.insecure.org/sploits/land.ip.DOS.html

[4] Ping of Death Attack. [Online]. Available: http://www.insecure.org/sploits/ping-o-death.html

[5] Teardrop Attack. [Online]. Available: http://support.microsoft.com/kb/q179129/

[6] "CAIDA's OC48 traces dataset." [Online]. Available: http://www.caida.org/data/passive/

[7] "CAIDA's skitter project." [Online]. Available: http://www.caida.org/tools/measurement/skitter/

[8] "Tor system." [Online]. Available: http://tor.eff.org

[9] "CAIDA's Internet Topology Data Kit #0304," Cooperative Association for Internet Data Analysis, San Diego Supercomputer Center (SDSC), University of California, San Diego (UCSD), 2003. [Online]. Available: http://www.caida.org/tools/measurement/skitter/router_topology/

[10] "Google click fraud settlement," Mar. 2006. [Online]. Available: http://googleblog.blogspot.com/pdf/lanes_google_final_order.pdf

[11] "Yahoo click fraud settlement," Jun. 2006. [Online]. Available: http://yhoo.client.shareholder.com/ReleaseDetail.cfm?ReleaseID=202354

[12] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan, "Approximating extent measures of points," *Journal of the ACM (JACM)*, vol. 51, no. 4, pp. 606–635, Jul. 2004.

[13] P. K. Agarwal, J. Matoušěk, and S. Suri, "Farthest neighbors, maximum spanning trees and related problems in higher dimensions," *Computational Geometry: Theory and Applications*, vol. 1, no. 4, pp. 189–201, Apr. 1992.

[14] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the 28th annual ACM symposium on Theory of computing (STOC 1996)*, Philadelphia, USA, May 1996, pp. 20–29.

[15] V. Anupam, A. Mayer, K. Nissim, B. Pinkas, and M. K. Reiter, "On the security of pay-per-click and other web advertising schemes," in *Proceedings of the 8th International Conference on World Wide Web (WWW 1999)*, Toronto, Canada, May 1999.

[16] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2004)*, Paris, France, Jun. 2004.

[17] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan, "Maintaining variance and k-medians over data stream windows," in *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2003)*, San Diego, USA, Jun. 2003, pp. 234–243.

[18] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2002)*, Madison, USA, Jun. 2002, pp. 1–16.

[19] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings of the 17th International Conference on Data Engineering (ICDE 2001)*, Heidelberg, Germany, Apr. 2001.

[20] A. Belenky and N. Ansari, "IP traceback with deterministic packet marking," *IEEE Communications Letters*, vol. 7, no. 4, pp. 162–164, Apr. 2003.

[21] S. M. Bellovin, "ICMP traceback messages," Internet Draft, 2000.

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

[23] A. Blum, D. Song, and S. Venkataraman, "Detection of interactive stepping stones: Algorithms and confidence bounds," in *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004)*, Sophia Antipolis, France, Sep. 2004.

[24] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," in *Proceedings of the 40th Annual Allerton Conference on Communication, Control, and Computing*, Monticello, USA, Oct. 2002, pp. 636–646.

[25] H. Burch and B. Cheswick, "Tracing anonymous packets to their approximate source," in *Proceedings of USENIX LISA 2000*, New Orleans, USA, Dec. 2000, pp. 319–327.

[26] C. Busch and S. Tirthapura, "A deterministic algorithm for summarizing asynchronous streams over sliding windows," in *Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS 2007)*, Aachen, Germany, Feb. 2007.

[27] T. M. Chan, "Faster core-set constructions and data stream algorithms in fixed dimensions," in *Proceedings of the 20th Annual Symposium on Computational Geometry (SoCG 2004)*, New York, USA, Jun. 2004, pp. 152–159.

[28] T. M. Chan and B. S. Sadjad, "Geometric optimization problems over sliding windows," in *Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC 2004)*, ser. Lecture Notes in Computer Science, vol. 3341. Hong Kong, China: Springer, Dec. 2004, pp. 246–258.

[29] S. Chang, L. Zhang, Y. Guan, and T. E. Daniels, "A framework for P2P botnets," in *Proceedings of the 2009 International Conference on Communications and Mobile Computing (CMC 2009)*, Kunming, China, Jan. 2009.

[30] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming (ICALP 2002)*, Malaga, Spain, Jul. 2002.

[31] B. Cheswick, H. Burch, and S. Branigan, "Mapping and visualizing the Internet," in *Proceedings of 2000 USENIX Annual Technical Conference*, San Diego, USA, Jun. 2000.

[32] K. L. Clarkson and P. W. Shor, "Applications of random sampling in computational geometry, II," *Discrete & Computational Geometry*, vol. 4, no. 1, pp. 387–421, Dec. 1989.

[33] G. Cormode and S. Muthukrishnan, "Radial histograms for spatial streams," DIMACS, Tech. Rep. 2003-11, 2003.

[34] ——, "What's hot and what's not: Tracking most frequent items dynamically," in *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2003)*, San Diego, USA, Jun. 2003, pp. 296–306.

[35] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," in *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, San Francisco, USA, Jan. 2002, pp. 635–644.

[36] D. Dean, M. Franklin, and A. Stubblefield, "An algebraic approach to IP traceback," *Information and System Security*, vol. 5, no. 2, pp. 119–137, 2002.

[37] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002)*, Rome, Italy, Sep. 2002, pp. 348–360.

[38] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable bloom filters," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD 2006)*, Chicago, USA, Jun. 2006.

[39] D. L. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford, "Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay," in *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Switzerland, Oct. 2002.

[40] eMarketer, "Online ad spending to total $19.5 billion in 2007," Feb. 2007. [Online]. Available: http://www.emarketer.com/Article.aspx?1004635

[41] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of ACM SIGCOMM 2002*, Pittsburgh, USA, Aug. 2002.

[42] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proceedings of the Internet Measurement Conference (IMC 2003)*, Miami, USA, Oct. 2003.

[43] 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.

[44] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently," in *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB 1998)*, New York, USA, Aug. 1998.

[45] J. Feigenbaum, S. Kannan, and J. Zhang, "Computing diameter in the streaming and sliding-window models," *Algorithmica*, vol. 41, no. 1, pp. 25–41, 2004.

[46] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[47] M. Gandhi, M. Jakobsson, and J. Ratkiewicz, "Badvertisements: Stealthy click-fraud with unwitting accessories," *Anti-Phishing and Online Fraud, Part I Journal of Digital Forensic Practice*, vol. 1, pp. 131–142, Nov. 2006.

[48] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book*. Prentice Hall, 2002.

[49] P. B. Gibbons and Y. Matias, "New sampling-based summary statistics for improving approximate query answers," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD 1998)*, Seattle, USA, Jun. 1998, pp. 331–342.

[50] P. B. Gibbons and S. Tirthapura, "Distributed streams algorithms for sliding windows," in *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, Winnipeg, Manitoba, Canada, Aug. 2002.

[51] L. Golab, D. DeHaan, E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Identifying frequent items in sliding windows over on-line packet streams," in *Proceedings of the Internet Measurement Conference (IMC 2003)*, Miami, USA, Oct. 2003.

[52] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson, "2005 CSI/FBI computer crime and security survey," 2005. [Online]. Available: http://www.usdoj.gov/criminal/cybercrime/CSI_FBI.htm

[53] R. Govindan and H. Tangmunarunkit, "Heuristics for Internet map discovery," in *Proceedings of IEEE INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000, pp. 1371–1380.

[54] Y. Guan and L. Zhang, "Attack traceback and attribution," in *Wiley Handbook of Science and Technology for Homeland Security*. Editor: J. G. Voeller, Wiley-Interscience.

[55] ——, "Detecting click fraud in pay-per-click streams of online advertising networks," US Patent Application, No. 12/187,055, Aug. 2008.

[56] M. R. Henzinger, P. Raghavan, and S. Rajagopalan, "Computing on data streams," Digital Systems Research Center, Palo Alto, USA, Tech. Rep. SRC Technical Note 1998-011, May 1998.

[57] J. Hershberger and S. Suri, "Adaptive sampling for geometric problems over data streams," in *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2004)*, Paris, France, Jun. 2004.

[58] B. Huffaker, D. Plummer, D. Moore, and k claffy, "Topology discovery by active probing," in *Proceedings of the 2002 Symposium on Applications and the Internet (SAINT 2002)*, Nara, Japan, Jan. 2002.

[59] P. Indyk, "Better algorithms for high-dimensional proximity problems via asymmetric embeddings," in *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, Baltimore, USA, Jan. 2003, pp. 539–545.

[60] Internet Advertising Bureau of United Kingdom, "IAB online adspend study," 2006. [Online]. Available: http://www.iabuk.net/en/1/iabknowledgebankadspend.html

[61] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Transactions on Database Systems (TODS)*, vol. 28, no. 1, pp. 51–55, Mar. 2003.

[62] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, China, Aug. 2002.

[63] H. T. Kung, F. Luccio, and F. P. Preparata, "On finding the maxima of a set of vectors," *Journal of the ACM (JACM)*, vol. 22, no. 4, pp. 469–476, Oct. 1975.

[64] L. Lee and H. Ting, "Maintaining significant stream statistics over sliding windows," in *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, Miami, USA, Jan. 2006.

[65] ——, "A simpler and more efficient deterministic scheme for finding frequent items over sliding windows," in *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2006)*, Chicago, USA, Jun. 2006.

[66] J. Li, M. Sung, J. Xu, and L. Li, "Large-scale IP traceback in high-speed internet: Practical techniques and theoretical foundation," in *Proceedings of 2004 IEEE Symposium on Security and Privacy*, Oakland, USA, May 2004.

[67] X. Lin, H. Lu, J. Xu, and J. X. Yu, "Continuously maintaining quantile summaries of the most recent N elements over a data stream," in *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, Boston, USA, Mar. 2004.

[68] X. Lin, Y. Yuan, W. Wang, and H. Lu, "Stabbing the sky: Efficient skyline computation over sliding windows," in *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, Tokyo, Japan, Apr. 2005.

[69] C. Lynn, W. Milliken, and W. T. Strayer, "SPIE memory requirements reduction," BBN Technologies, Tech. Rep. BBN REPORT-8385, Dec. 2003.

[70] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston, "Finding (recently) frequent items in distributed data streams," in *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, Tokyo, Japan, Apr. 2005.

[71] A. Mankin, D. Massey, C.-L. Wu, S. F. Wu, and L. Zhang, "On design and evaluation of "Intention-Driven" ICMP traceback," in *Proceedings of the 10th IEEE International Conference on Computer Communications and Networks (ICCCN 2001)*, Scotsdale, USA, Oct. 2001.

[72] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, China, Aug. 2002.

[73] C. C. Mann, "How click fraud could swallow the Internet?" Jan. 2006. [Online]. Available: http://www.wired.com/wired/archive/14.01/fraud.html

[74] S. Matsuda, T. Baba, A. Hayakawa, and T. Nakamura, "Design and implementation of unauthorized access tracing system," in *Proceedings of the 2002 Symposium on Applications and the Internet (SAINT 2002)*, Nara, Japan, Jan. 2002.

[75] A. Metwally, D. Agrawal, and A. E. Abbadi, "Duplicate detection in click streams," in *Proceedings of the 14th WWW International World Wide Web Conference (WWW 2005)*, Chiba, Japan, May 2005.

[76] ——, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of the 10th International Conference on Database Theory (ICDT 2005)*, Edinburgh, Scotland, Jan. 2005.

[77] ——, "Using association rules for fraud detection in web advertising networks," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, Trondheim, Norway, Aug. 2005.

[78] ——, "DETECTIVES: DETEcting Coalition hiT Inflation attacks in adVertising nEtworks Streams," in *Proceedings of the 16th WWW International World Wide Web Conference (WWW 2007)*, Alberta, Canada, May 2007.

[79] A. Metwally, D. Agrawal, A. E. Abbadi, and Q. Zheng, "On hit inflation techniques and detection in streams of web advertising networks," in *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS 2007)*, Toronto, Canada, Jun. 2007.

[80] J. Misra and D. Gries, "Finding repeated elements," *Science of Computer Programming*, vol. 2, no. 2, pp. 143–152, Nov. 1982.

[81] M. Mitzenmacher, "Compressed Bloom filters," *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604–612, Oct. 2002.

[82] S. Muthukrishnan, "Data streams: Algorithms and applications," Rutgers University, Piscataway, USA, Tech. Rep., 2003.

[83] Outsell Survey, "Hot topics: Click fraud reaches $1.3 billion, dictates end of "don't ask, don't tell" era," Jun. 2006. [Online]. Available: http://www.outsellinc.com/store/products/243

[84] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*, San Diego, USA, Jun. 2003.

[85] K. Park and H. Lee, "On the effectiveness of probabilistic packet marking for IP traceback under denial of service attack," in *Proceedings of IEEE INFOCOM 2001*, Anchorage, USA, Apr. 2001, pp. 338–347.

[86] Y. Peng, L. Zhang, and Y. Guan, "Combating malicious-script-generating click frauds," submitted to the 30th IEEE Symposium on Security and Privacy (S&P 2009).

[87] ——, "An effective fraud detection method for Internet auction systems," in *Proceedings of the 5th Annual IFIP Working Group 11.9 International Conference on Digital Forensics*, Orlando, USA, Jan. 2009.

[88] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. New York, USA: Springer, 1985.

[89] E. A. Ramos, "An optimal deterministic algorithm for computing the diameter of a three-dimensional point set," *Discrete & Computational Geometry*, vol. 26, no. 2, pp. 233–244, 2001.

[90] M. K. Reiter, V. Anupam, and A. Mayer, "Detecting hit shaving in click-through payment schemes," in *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, Boston, USA, Aug. 1998, pp. 155–166.

[91] R. Richardson, "2008 CSI computer crime & security survey," 2008. [Online]. Available: http://www.gocsi.com/forms/csi_survey.jhtml

[92] T. J. Richardson, "Approximation of planar convex sets from hyperplane probes," *Discrete & Computational Geometry*, vol. 18, no. 2, pp. 151–177, 1997.

[93] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical network support for IP traceback," in *Proceedings of ACM SIGCOMM 2000*, Stockholm, Sweden, Aug. 2000, pp. 295–306.

[94] K. Shanmugasundaram, H. Brönnimann, and N. Memon, "Payload attribution via hierarchical Bloom filters," in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS 2004)*, Washington DC, USA, Oct. 2004.

[95] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer, "Single-packet IP traceback," *IEEE/ACM Transactions on Networking*, vol. 10, no. 6, pp. 721–734, Dec. 2002.

[96] D. Song and A. Perrig, "Advanced and authenticated marking schemes for IP traceback," in *Proceedings of IEEE INFOCOM 2001*, Anchorage, USA, Apr. 2001.

[97] E. H. Spafford, "OPUS: Preventing weak password choices," *Computers & Security*, vol. 11, no. 3, pp. 273–278, May 1992.

[98] S. Staniford-Chen and L. T. Heberlein, "Holding intruders accountable on the internet," in *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, Oakland, USA, May 1995.

[99] R. Stone, "Centertrack: An IP overlay network for tracking DoS floods," in *Proceedings of the 9th USENIX Security Symposium*, Denver, USA, Aug. 2000, pp. 199–212.

[100] W. T. Strayer, C. E. Jones, I. Castineyra, J. B. Levin, and R. R. Hain, "An integrated architecture for attack attribution," BBN Technologies, Tech. Rep. BBN REPORT-8384, Dec. 2003.

[101] W. T. Strayer, C. E. Jones, F. Tchakountio, and R. R. Hain, "SPIE-IPv6: Single IPv6 packet traceback," in *Proceedings of the 29th IEEE Local Computer Networks Conference (LCN 2004)*, Tampa, USA, Nov. 2004.

[102] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*, Roma, Italy, Sep. 2001.

[103] Y. Tao and D. Papadias, "Maintaining sliding window skylines on data streams," *IEEE Transactions on Knowledge and Data Engineering archive (TKDE)*, vol. 18, no. 3, pp. 377–391, Mar. 2006.

[104] S. Tirthapura, B. Xu, and C. Busch, "Sketching asynchronous streams over sliding windows," in *Proceedings of the 25th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2006)*, Denver, USA, Jul. 2006.

[105] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum, "New streaming algorithms for fast detection of superspreaders," in *Proceedings of Network and Distributed Systems Security Symposium (NDSS 2005)*, San Diego, USA, Feb. 2005.

[106] X. Wang and D. S. Reeves, "Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, Washington DC, USA, Oct. 2003.

[107] X. Wang, D. S. Reeves, and S. F. Wu, "Inter-packet delay based correlation for tracing encrypted connections through stepping stones," in *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS 2002)*, Zurich, Switzerland, Oct. 2002.

[108] X. Wang, D. S. Reeves, S. F. Wu, and J. Yuill, "Sleepy watermark tracing: An active network-based intrusion response framework," in *Proceedings of the 16th International Conference on Information Security (IFIP/Sec 2001)*, Paris, France, Jun. 2001.

[109] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 2, pp. 208–229, Jun. 1990.

[110] S. F. Wu, L. Zhang, D. Massey, and A. Mankin, "Intention-Driven ICMP trace-back," Internet Draft, 2001.

[111] J. Xin, L. Zhang, B. Aswegan, J. Dickerson, J. Dickerson, T. Daniels, and Y. Guan, "A testbed for evaluation and analysis of stepping stone attack attribution techniques," in *Proceedings of 2nd International IEEE/Create-Net Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2006)*, Barcelona, Spain, Mar. 2006.

[112] J. Xin, L. Zhang, T. Daniels, J. Dickerson, and Y. Guan, "A merge/split robust scheme for attributing stepping stone attacks in real-world networked systems," to be submitted.

[113] K. Yoda and H. Etoh, "Finding a connection chain for tracing intruders," in *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, Toulouse, France, Oct. 2000.

[114] L. Zhang and Y. Guan, "TOPO: A topology-aware single packet attack traceback scheme," in *Proceedings of the 2nd IEEE Communications Society/CreateNet International Conference on Security and Privacy in Communication Networks (SecureComm 2006)*, Baltimore, USA, Aug. 2006.

[115] ——, "Variance estimation over sliding windows," in *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2007)*, Beijing, China, Jun. 2007.

[116] ——, "Detecting click fraud in pay-per-click streams of online advertising networks," in *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS 2008)*, Beijing, China, Jun. 2008.

[117] ——, "Frequency estimation over sliding windows," in *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*, Cancun, Mexico, Apr. 2008.

[118] L. Zhang, A. G. Persaud, A. Johnson, and Y. Guan, "Detection of stepping stone attack under delay and chaff perturbations," in *Proceedings of the 25th IEEE International Performance Computing and Communications Conference (IPCCC 2006)*, Phoenix, USA, Apr. 2006.

[119] Y. Zhang and V. Paxson, "Detecting stepping stones," in *Proceedings of the 9th USENIX Security Symposium*, Denver, USA, Aug. 2000, pp. 171–184.

[120] Q. Zhao, A. Kumar, and J. Xu, "Joint streaming and sampling techniques for accurate identification of super sources/destinations," in *Proceedings of the Internet Measurement Conference (IMC 2005)*, Berkeley, USA, Oct. 2005, pp. 77–90.

[121] Y. Zhu and D. Shasha, "StatStream: Statistical monitoring of thousands of data streams in real time," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, China, Aug. 2002.