

Detecting Exploit Patterns from Network Flow Streams

Research Proposal for Ph.D. Preliminary Examination

Bibudh Lahiri

Department of Electrical and Computer Engineering

Iowa State University

bibudh@iastate.edu

Thesis Advisor: Dr. Srikanta Tirthapura

PoS Committee: Dr. Srinivas Aluru

Dr. Soma Chaudhuri

Dr. Daji Qiao

Dr. Aditya Ramamoorthy

Date: 10:00 am-12:00 pm, Nov 14, 2008

Place: 2222 Coover Hall

Abstract

An Intrusion Detection System (IDS) is a piece of software and/or hardware designed to detect unwanted attempts at accessing, manipulating, and disabling of computer systems, through a network such as the Internet. Network-based Intrusion Detection Systems (NIDS) try to detect malicious activities by monitoring network traffic. Research on network traffic measurement has identified various patterns that the typical exploits on today's Internet exhibit. The goal of our research is to devise single-pass (online) data stream algorithms for detecting these patterns from network traffic flow data, using a workspace that is much smaller than the size of the traffic flow. We aim to design algorithms with a provable guarantee on the space and time requirements and the degree of approximation in the estimates returned.

Contents

1	Introduction	5
1.1	Challenges in Network Monitoring	5
1.2	Data Streams: Model and Algorithms	6
1.3	Denial of Service attacks	10
1.4	Port Scans	12
1.5	NIDS: The Current State of the Art	14
1.6	Our Objective	15
1.7	The Research Plan	15
1.8	Organization	16
2	Identification of Persistent Exploit Sources	17
2.1	Motivation	17
2.2	Problem Formulation	20
3	Statistics of Frequent Items	21
3.1	Motivation	21
3.2	Related Work	23
3.3	Problem Formulation	25
3.4	Progress: An Approximation Solution	27
4	Identifying Heavy-Hitters from Distributed Datasets	28
4.1	Motivation	28

4.2	Related Work	28
4.3	Problem Formulation	30
4.4	Research Results	31
4.4.1	Absolute Threshold	31
4.4.2	Relative Threshold	32
5	Conclusion	34

Chapter 1

Introduction

An Intrusion Detection System (IDS) is a piece of software or hardware designed to detect unwanted attempts at accessing, manipulating and disabling of computer systems through a network such as the Internet. An Intrusion Detection System can be network-based, protocol-based or host-based. Network-based Intrusion Detection Systems (NIDS), e.g., Snort [45], Bro [43] or NSM [28], try to detect malicious activity such as Denial of Service (DoS) attacks [40], port scans [46] or even attempts to crack into computers by monitoring network traffic. The network traffic measurement researchers have identified various patterns that the typical exploits on today's Internet (e.g., DoS attacks, port scans, worms) exhibit [48, 49]. However, there has not been any significant attempt, so far, to design algorithms - with theoretical guarantees on the space and/or time requirement, or the extent of approximation - for detecting these known exploit patterns from network traffic flow data and applying them in NIDSs. We observe that data stream algorithms, which compute various aggregates from massive data streams online and in small space, can be applied to detect such exploit patterns from network flow data. The goals of this research are (1) formalization of the notion of exploit patterns and (2) design and analysis of efficient algorithms for detecting these patterns from network flow streams.

1.1 Challenges in Network Monitoring

The Internet consists of routers connected to each other that forward IP packets. Traffic at the routers may be viewed at several levels of abstraction [42, 23].

1. **Packet logs:** Each TCP segment has a header that has source and destination ports, and the IP packet that the TCP segment envelops contains the source and destination IP addresses in its header. This can be collected at switches, routers or network taps by tools like tcpdump [1] or wireshark [2].
2. **Flow logs:** Each flow is a collection of packets with same values for certain key attributes such as the source and destination IP addresses. For each flow, the log contains cumulative information about number of bytes and packets sent, start and end times, protocol types etc. This is typically collected at border routers or taps by tools like FlowScan [44], YAF [3] or Argus [4].
3. **Traffic counters:** A traffic counter keeps track of the number of bytes sent over each link every few minutes. An example is MRTG logs [5], which record the volume of traffic through SNMP-enabled devices.

It is most valuable to analyze flow logs and packet logs, because they contain maximum information. However, this, at the same time, implies that we should be prepared to deal with enormous volume of data. For example, the backbone of a typical regional ISP today is a OC-48 network line with a transmission speed of 2.5 Gbits/sec, of which the packet overhead part alone is 83 Mbits/s! Given the *volume* and the *short-lived* nature of this data, it is almost impossible to store these data on a hard disk, not to mention in a main memory, and any analysis on the data has to be performed *online*, i.e., accepting the fact that we get to see each data item only once.

1.2 Data Streams: Model and Algorithms

A *data stream* is an abstract model for applications where data is generated continuously (e.g., stock quotes, network flows, call records in a telephone exchange, high-energy particle physics experiments).

Definition 1.2.1 A *data stream* $A = (a_1, a_2, \dots, a_m)$ is a *sequence of elements*, where each a_i is a member of $[n] = \{1, 2, \dots, n\}$.

The sheer volume and transience of data streams forces us to perform any computation on the data in a single pass and using limited memory. These constraints have motivated the emergence of a class of data structures called *sketches*, defined as follows:

Definition 1.2.2 A sketch is a data structure with the following properties:

- **Property 1.2.1** A sketch requires small space, typically polylogarithmic in the size of the stream, or the size of a subset of the stream we are interested in
- **Property 1.2.2** It can be updated, in constant or polylogarithmic time, as the elements of the stream are received
- **Property 1.2.3** The aggregate that we want to compute on the stream can be computed approximately based on the sketch
- **Property 1.2.4** Some applications, with distributed streams (to be defined later), requires the following property: if a separate sketch is maintained for each of two or more streams, then the combination of the sketches should be able to answer the desired aggregate on the union of the streams, with guaranteed accuracy

A sketch can be a simple uniform or weighted random sample of the stream elements, or a projection along random vectors, or any other transformation that satisfies the above properties.

We now present some basic research findings in data streams. Let $m_i = |\{j : a_j = i\}|$ denote the number of occurrences of i in the sequence A . For each $k > 0$, a useful statistics of the sequence is the k^{th} frequency moment, defined as $F_k = \sum_{i=1}^n m_i^k$. In particular, F_0 is the number of distinct elements in the sequence, $F_1 (= m)$ is the length of the sequence, and F_2 is known as the “surprise index”.

In a seminal work, Alon *et al* [6] analyzed the space complexity of computing the frequency moments. They presented lower bounds showing that an exact computation of F_k , or even an accurate deterministic approximation of it, requires $\Omega(n)$ space, in the worst case. However, a randomized approximation to F_k (for $k \geq 2$) can be found as follows. First, choose a random element a_p from A . Then maintain the count $X = |\{q : q \geq p, a_q = a_p\}|$. In other words, count the number of reoccurrences of the element a_p in the portion of the stream that succeeds a_p (including a_p). Then, the random variable $Y = m[X^k - (X - 1)^k]$ is an unbiased estimator of F_k , i.e. $E[Y] = F_k$. Further, it can also be shown that the variance of Y is small. They proved that F_0 , F_1 and F_2 can be approximated in logarithmic space, whereas the approximation

of F_k for $k \geq 6$ requires $n^{\Omega(1)}$ space. Of these, the problem of estimating F_0 has drawn significant attention of the researchers, and has been addressed by Flajolet and Martin [22] and Gibbons and Tirthapura [24].

The problem of identifying the frequently occurring items [15, 38, 39] (often termed “heavy-hitters”) from a stream has also been studied quite thoroughly. For any user-input threshold $\phi \in (0, 1)$, Misra and Gries [39] came up with a deterministic algorithm to find the data items that occur more than ϕm times in an array of size m . Their algorithm required $O(m \log \frac{1}{\phi})$ time and $O(\frac{1}{\phi})$ space for the sketch. The problem with their algorithm was that it was *two-pass* - the first pass could identify the candidates for frequent items, and kept track of the counts of these elements; and in the second pass, one had to eliminate, from these candidates, the ones that were not actually frequent. However, with minor modifications of the original algorithm, and a little sacrifice in precision, it is possible to come up with a *single-pass, approximate* variant of the algorithm that provides the following approximation guarantees, for some user-input threshold ϕ and approximation error $\epsilon < \phi$ (note that for an *online* algorithm, m is the number of elements received *so far*):

- All items whose frequencies exceed ϕm are output. There are no false negatives.
- No item with frequency less than $(\phi - \epsilon)m$ is output.
- Estimated frequencies are less than true frequencies by at most ϵm .

The algorithm is simple: it maintains a hashtable (of size at most $\frac{1}{\epsilon}$) of (value, count) pairs. On receiving each item a_i , we check whether the value a_i is already in the hashtable. If it exists, we increment its count by 1; otherwise, we add the pair $(a_i, 1)$ to the hashtable. Now, if adding a new pair to the hashtable makes its size exceed $\frac{1}{\epsilon}$, then for each of the (value, count) pairs in the hashtable, we decrement the count by one; and throw away any value whose count falls to zero after decrement. Note that this ensures at least the element which was most recently added (with a count of one) would get discarded, so the size of the hashtable, after processing all pairs, would come down to $\frac{1}{\epsilon}$ or less. Thus, the space requirement of this algorithm is $O(\frac{1}{\epsilon})$.

In data stream applications, often, the more recent an item is, the more is our interest in it. So, a popular model for studying data streams is the *sliding-window* model, where we focus on computing the aggregates on the last N items of the stream, using $o(N)$ space. Datar *et al* [18] solved the *basic counting* problem in

the sliding-window model: given a stream of bits, they came up with an ϵ -approximate algorithm for counting the number of 1's among the last N bits, using $O(\frac{1}{\epsilon} \log^2 N)$ bits of memory. Their algorithm processed each item in $O(1)$ amortized and $O(\log N)$ worst-case time. They also extended their algorithm to maintain the sum of last N elements, with a relative error of at most ϵ , in a stream of positive integers in the range $[0 \dots R]$. This algorithm needed $O(\frac{1}{\epsilon}(\log N + \log R)(\log N))$ memory bits. The arrival of each new element was processed in $O(\frac{\log N}{\log R})$ amortized time and $O(\log N + \log R)$ worst-case time. Gibbons and Tirthapura [25] came up with improved results for the same problem, using a novel data structure called the *wave*. For the basic counting problem, they improved the per-item processing time to $O(1)$ in worst case. For the sum problem, they improved the worst-case per-item processing time to $O(1)$.

With the emergence of huge networks, today's applications often collect data not from a single source, but from distributed sources. In such a scenario, we are interested in computing aggregates over the *union* of the streams emerging from different sources. As we have already discussed, network monitoring devices observe streams of packets. Each device has a small workspace in which to store information on its observed stream, and the contents are periodically sent to a central data analyzer, in order to compute aggregated statistics on the streams. Some existing network monitoring tools, e.g., Lucent's InterpretNet and some products implementing Cisco's NetFlow protocol, use this mechanism for traffic monitoring.

As Gibbons and Tirthapura [24] pointed out, there is a subtle difference between the distributed streams model and the *merged* streams model. In the latter model, there is only one party who observes both streams, and the streams are interleaved in an arbitrary order by an adversary. In the distributed streams model, each party observes its own stream, and computes the sketch on it. The sketches have the property that when combined, they can give approximate answers to queries over the unions of all the streams. Gibbons and Tirthapura [24] showed that for $t > 2$ streams and for any function f , the deterministic merged stream complexity (i.e., space bound) is within a factor of t of the deterministic t -party distributed stream complexity. It followed that deterministic merged streams algorithms can be designed assuming that the streams are not interleaved, at a penalty of at most t .

Our study of the intrusion detection literature reveals that most of the NIDS tools, in practice, work by checking the packet payloads for signatures of well-known attacks. This is not a very scalable method as the attacker can evade detection with minor changes in the signature, but can still amount the same damage to the victim. Also, the signature set can grow very large, e.g., the signature set of Snort contains 1,715 distinct signatures, of which 1,273 are enabled by default. Matching these signatures sequentially with the packet payload is very expensive.

We found that in the area of network traffic measurement, there has been systematic study of real network traffic data to identify various patterns of the typical exploits on today's Internet (e.g., DoS attacks, port scans, worms) [48, 49, 50]. Xu *et al* [48, 49] applied data mining and information-theoretic techniques to automatically extract useful information from largely unstructured data. This shows that exploit patterns can be detected by computing summary statistics over large volumes of data, rather than inspecting the payload of each and every packet for an exact or partial signature match. We observed that DoS attacks [40] and port scans [46] are two types of attacks on the Internet where data-driven approaches can play major roles in identifying potential threats, and hence discuss them in detail in the following sections.

1.3 Denial of Service attacks

A DoS attack is an attempt to make a computer resource (e.g., a website or a database server) unavailable to its intended users. The methods generally involve saturating the target machine with external communications requests, such that it cannot respond to legitimate traffic, or responds so slowly as to be rendered effectively unavailable; or, saturating some other resource, such as a system buffer.

In February 2000, a series of massive DoS attacks incapacitated several high-visibility Internet e-commerce sites, including Yahoo, Ebay, and E*trade. Next, in January 2001, the name server infrastructure of Microsoft was disabled by a similar assault. Many other domestic and foreign sites have also been victims, ranging from smaller commercial sites, to educational institutions, public chat servers and government organizations. Moore *et al* [40] monitored a lightly utilized network, comprising 2^{24} distinct IP addresses, over a period of 25 days in February 2001, and observed 12,805 attacks on over 5,000 distinct Internet hosts

belonging to more than 2,000 different organizations during this period. The following few types of DoS attacks are most common:

SYN Flood: This is perhaps the most common and widely studied form of DoS attack. When a client attempts to start a TCP connection to a server, the client and server exchange a series of messages which normally runs like this:

1. The client requests a connection by sending a SYN (synchronize) message to the server.
2. The server acknowledges this request by sending SYN-ACK back to the client.
3. The client responds with an ACK, and the connection is established.

This is called the *TCP three-way handshake*, and is the foundation for every connection established using the TCP protocol. The SYN Flood attack works if a server allocates resources for the connection after receiving a SYN, but before it has received the ACK. The basic mechanism to launch a SYN flood is the following: if half-open connections bind resources on the server, it may be possible to take up all these resources by flooding the server with SYN messages. Once all resources set aside for half-open connections are reserved, no new connections (legitimate or not) can be made, resulting in denial of service. There are the two following methods to launch a SYN flood attack - both involve the server not receiving the ACK.

- A malicious client can skip sending this last ACK message.
- By falsifying the source IP address in the SYN, it can make the server send the SYN-ACK to the falsified IP address, and thus never receive the ACK.

UDP Flood: A UDP flood attack can be initiated by sending a large number of UDP packets to random ports on a remote host. Since the target host finds (with high probability) that no application is listening at that port, it replies with an “ICMP Destination Unreachable” packet. Thus, for a large number of UDP packets, the victim host is forced to send many ICMP packets, eventually leading it to be unreachable by other clients.

There are some other forms of DoS attacks (e.g., Smurf attack, Ping flood), the underlying idea behind all of them being the same: the attacker pretends to be a benign host, and initiates some form of communication request with the victim, the scale of the communication being large enough to eventually exhaust

all the system resources of the victim. A variant of DoS attacks worth mention here is the Distributed DoS (DDoS) attack, where instead of a single attacker, multiple compromised systems flood the bandwidth or resources of a targeted system. The attack can be initiated by a single attacker, who compromises other systems to launch an attack on the end victim(s) on a large scale. Note that if we treat the traffic flow of a single compromised system as a data stream, then the distributed streams model [24, 25] can be useful in mining patterns from the union of these streams.

A DoS attack, or at least an attempt to launch one, can be detected by applying heavy-hitter algorithms on traffic flow data. Where the frequent sources can be the potential attackers, the frequent destinations can be the potential victims. In Chapter 3, we discuss how we plan to extend the heavy-hitter problem to design more informative sketches for detecting the attackers and victims in a DoS attack under certain conditions. In Chapter 4, we will discuss some sketches we have already developed [36] for identifying heavy-hitters from the union of multiple data streams, which can be used for detecting DDoS attacks.

1.4 Port Scans

Some applications, running on specific ports, have some known vulnerabilities, and sometimes the network-based exploits take advantage of these vulnerabilities. The exploit traffic is hence directed towards this port, e.g., the W32/Blaster worm took advantage of a buffer overflow in the Microsoft DCOM RPC locator service, an application that runs on TCP port 135, to create a SYN flood. *Port scanning* is a technique to search a network host for *open* ports, i.e., ports where a deployed application with known vulnerabilities is looking for an incoming connection.

Jung *et al* [29] pointed out that a number of difficulties arise when we attempt to formulate an effective algorithm for detecting port scans. The first is that there is no crisp definition of the activity. e.g., an attempted HTTP connection to the main web server of a site is should not raise an alarm. However, whether a sweep through the entire address space looking for HTTP servers should concern us depends on what *intent* the sweep is being done with. e.g., some search engines not only follow embedded links but also scan ports in order to find web servers to index. In addition, some applications (e.g., SSH, some P2P and Windows

applications) have modes in which they scan in a benign attempt to gather information or locate servers. Ideally, we would like to separate out such benign use from overtly malicious one. We would note, however, that the question of whether scanning by search engines is benign will ultimately be a *policy* decision that will reflect the sites view of the desirability to have information about its servers publicly accessible.

Since we never know whether the intent behind a port scan is harmful or not, being able to detect port scans is important from a security perspective. If we are allowed to make multiple passes over data gathered from packet/flow logs, then port scans can surely be detected. However, detecting them online and using small space presents the usual challenge of network monitoring.

We present a classification of port scans from [46]:

1. **Vertical Scan:** A sequential or random scan of multiple ports of a single IP address from the same source in a given time window. These are usually an attempt to survey which of several well known vulnerabilities applies to this host.
2. **Horizontal Scan:** A scan from a single source of several machines in a subnet aimed at the same target port, i.e., the same vulnerability. In this case the attacker is searching for any machine that is running specific service and does not care about any single machine in particular.

Just like DoS attacks can be distributed ones, horizontal or vertical port scans can also be launched by multiple sources working in tandem (sometimes referred to as “Coordinated Scans” in the literature). However, horizontal or vertical port scans can be easily detected offline from packet/flow logs, unless the scan is a *stealth scan*. A stealth scan is initiated with a very low frequency to avoid detection. The key parameters in the definition of stealth scan include the maximum threshold and the minimum threshold for the average interscan distance. An average interscan distance below the minimum threshold indicates that the scan was not stealthy, i.e., not intended to evade NIDS systems. Two successive scans from the same source that are separated by more than the maximum interscan distance are considered to be unrelated or parts of different scanning episodes. We will discuss in Chapter 2 how we can formalize the notion of “persistence” of data items to detect sources that launch stealthy port scans.

1.5 NIDS: The Current State of the Art

Having discussed two major exploit patterns, DoS attacks and port scans, we now move on to discuss some of the measures that have been adopted so far to address these problems. Traditionally, intrusion detection tools are classified into two broad categories: *signature-based* and *anomaly-based* [27]. Signature-based NIDSs aim to detect well-known attacks as well as slight variations of them, by identifying the *signatures* that characterize these attacks. Due to its nature, a signature-based NIDS has *low false positives* but it is unable to detect any attacks that lie beyond its knowledge. An anomaly-based NIDS is designed to capture any deviations from the established profiles of users and the normal behavior patterns of systems. Although in principle, anomaly detection has the ability to detect new attacks, in practice this is far from easy. Anomaly detection has the potential to generate too many false alarms, and it is very time consuming and labor-expensive to sift true intrusions from the false alarms.

Historically, most signature-based NIDS tools function by detecting N or more events in a time window of T seconds. Network Security Monitor (NSM) [28] was the first NIDS to work on such algorithm. It had rules to detect any source IP address connecting to more than 15 distinct destination IP addresses within a given time window. Similarly, Snort [45], a tool developed later by Martin Roesch, checks whether a given source IP address connected to more than X number of ports or more than Y number of destination IP addresses within Z seconds, where X , Y , Z are configurable parameters. However, Bro [43], another popular NIDS, worked on the observation that *failed* connection attempts are *better* indicators for identifying port scans. Since scanners have little knowledge of network topology and system configuration, they are likely to often choose an IP address or port that is not active. The algorithm provided by Bro treated connections differently depending on their services (application protocols). For connections using a service specified in a configurable list (e.g., HTTP, SSH, SMTP etc), Bro only performs bookkeeping if the connection attempt failed (was either unanswered, or elicited a TCP RST response). For others, it considers all connections, whether or not they failed. It then tallies the number of distinct destination addresses to which such connections (attempts) were made. If the number reaches a configurable parameter N , then Bro flags the source address as a scanner.

The developers of the NIDS tools focussed on building full-fledged applications that can monitor network traffic, generate logs and report incidents (the definition of “incident” depends on the policy of the user, and is configurable within the tool). While these tools are lightweight and (mostly) open-source, and some detailed documentation regarding their technical architectures are available [21], it seems that the overall approach is *not* very *scalable*. Most NIDSs look for an *exact* match for a signature, which is a sequence of bytes, in the payloads of packets. However, when using tight signatures, the matcher has no capability to detect attacks other than those for which it has explicit signatures; the matcher will in general completely miss novel attacks, which, unfortunately, continue to be developed at a brisk pace. Also, the total number of patterns contained in the signature set of a NIDS can be quite large, hence matching them sequentially is time-wise expensive. Although Gonzalez and Paxson [26] came up with some sampling-based modifications of Bro to identify the heavy-hitters from the traffic streams, there is no published literature on their algorithms with theoretical guarantees, and there is certainly scope for identifying more such aggregates from traffic streams.

1.6 Our Objective

The goal of our research is to apply tools from data stream processing in identifying network intrusions by detecting exploit patterns in an online fashion. Note that network exploits may not always be detectable in the way the existing NIDSs try to detect them, e.g., by matching packet contents with existing signatures in the forms of hard-coded strings or even regular expressions. Sometimes, attacks can be detectable only by looking at the traffic streams at an aggregate level, rather than a per-packet basis. That calls for the design of efficient algorithms which can mine apparently hidden patterns from data streams in an online fashion and using limited memory. This is where data stream processing fits in.

1.7 The Research Plan

We plan to contribute towards the development of network-based intrusion detection techniques in the following way:

- Learn more about the typical patterns that network flow data exhibit in presence of exploits, by study-

ing existing literature from the area of network measurement

- Design (or combine existing) data stream sketching techniques to detect these patterns from network flow data
- Simulate the patterns synthetically and then test our algorithms on the generated data

1.8 Organization

We have explained the importance of developing efficient algorithms for network monitoring, and why we chose data stream algorithms as our tool. In the later sections, we present a detailed description of our research plan, and the progress we have made so far. We will present some relevant findings from the network measurement literature that motivated our work. Then, we will discuss the problems that we want to work on, and our progress in this intended area of research till date.

Chapter 2

Identification of Persistent Exploit Sources

2.1 Motivation

In Section 1.2, we discussed some popular data stream aggregates. Most of them, particularly the heavy-hitters and the frequency moments, are primarily concerned about the frequency distributions of data values, and not their temporal distributions. In other words, we only queried how many times a data item appeared in a stream or in a subset of the stream, but not in what *order* they appeared. However, the order of appearance may become relevant in certain cases, as we point out in the following paragraphs.

Using the packet traces collected from an OC48 backbone link during a 24-hour period, Xu *et al.* [49] pointed out a common behavior that the exploit sources are often *persistent*, i.e., they *continuously* send exploit traffic over a span of time and then disappear. Motivated by this observation, they considered finding the *persistent* source IP addresses from a network traffic data stream. They divided the 24-hour window into 5-minute slots, (so that there were $12 \cdot 24 = 288$ slots in total) and defined persistence as the number of *consecutive* five-minute slots over the total number of periods that a source sent significant amount of exploit traffic. So, if a source sent exploit traffic over the time slots 1, 2, 3, 4, 5, 10, 19, 250, 251, 252, 253 and 254, then as per their definition, since the intervals 1-5 and 250-254 (each of length 5) are continuous ones, and there are 12 slots in total, its persistence would be $\frac{5+5}{12} = 83.33\%$. [49] revealed that 60% of the *frequent* sources (a source was termed *frequent* if it sent *exploit* traffic in two or more of the 288 time

slots) had a persistence of 100%. Had they divided the 24-hour window to even smaller intervals, then fewer sources would have been likely to have a persistence of 100%.

We propose an alternate, and perhaps more general definition of persistence. Of course, the precise mathematical definition would depend on what *characteristic* of an exploit source we want to capture by our notion of persistence. Suppose we have 10 time slots, numbered 1 to 10, and two sources s_1 and s_2 . Suppose further s_1 sends data in the slots 1, 2, 3, 4 and 5; and s_2 sends data in the slots 1, 3, 5, 7 and 9. As per the definition of Xu *et al.* [49], the persistence of s_1 and s_2 would be 100% and 0% respectively, since s_1 sends data in all consecutive time-slots, but s_2 does not send its data in consecutive time-slots. However, we want to come up with a formal definition of persistence where the difference between s_1 and s_2 would not be this extreme; because, even though s_2 does not send the data in consecutive time-slots unlike s_1 , it turns up “pretty regularly” in the entire range of 10 slots; so it should not be altogether neglected, unlike in [49]. We show how an alternate definition of persistence can be offered on top of their data model, as follows. If a source s sends data in n slots $t_1, t_2, t_3, \dots, t_n$, then we define its persistence (p_s) as the sum of the reciprocals of all the inter-arrival times, i.e.,

$$p_s = \sum_{j=1}^{n-1} \frac{1}{t_{j+1} - t_j}$$

As per this definition, in the above example, the persistence of the sources s_1 and s_2 are respectively 4 and 2, so s_1 is twice as persistent as s_2 . Note that if we have a third source s_3 which sends data at timeslots 1,5 and 10, then its persistence would be $\frac{9}{20} < 0.5$. As per our definition, the persistence of a source can be as large as $T - 1$, where T is the total number of timeslots, and the source sends data in all of them. A source would be least persistence if it never sends data, and we trivially define its persistence as 0.

According to our definitions of persistence, the sources which send data over consecutive slots will be considered as most persistent; however, the sources which do not send data over consecutive slots, but show up every now and then, will not be ignored altogether. This definition of persistence can be useful in several ways. First, it can help us detect sources which conduct a stealthy port scan (refer to Section 1.4 for definition). The sources which perform a stealthy port scan would most likely never send data at consecutive

intervals, in order to evade detection by the NIDS. However, to ensure that they get to scan enough number of ports or destination IP addresses, they would keep on sending data intermittently, so their persistence values will not be too large, but will not be too small either. Second, if we want to identify some exploit sources that are persistence but “volatile”, i.e., they do whatever damage they wish to do within a short period of time and then disappear, then their persistence will be quite high in that interval, and we can identify them by estimating their persistence as per this definition and noting that it exceeds the cutoff.

Note that persistence is an aspect of the stream data items which is quite different from their frequencies. As our above example with the sources s_1 and s_2 reveals, two sources can have the same frequency, yet their persistence can be very different. The notion of persistence is also very different from the notion of recency, which has been captured by the sliding window model [18, 25] and by the more generic time-decay models [17]. As we have already discussed in Section 1.2, in the sliding window model, only the last N items of a stream are considered for any aggregate computation, where N is the user-given window size. Tirthapura *et al* [17] considered a more generic time-decay model, where the stream element a_i appeared as a 3-tuple of value, weight and timestamp, i.e., $a_i = (v_i, w_i, t_i)$. Now, at the current time c , the age of a_i is the time elapsed since its appearance, which is $c - t_i$. In this model, the frequency or weight of a_i at time c was taken to be $f(c - t_i)$, where f is a monotonically decreasing function, e.g., $f(c - t_i)$ can be $e^{-(c-t_i)}$. The intuition was that the weight (or importance) of an element decayed gradually with time. Tirthapura *et al* [17] proposed algorithms for computing various aggregates on the stream under this time-decay model. Interestingly, as per our definition of persistence, for some time-decay models, *a less persistent item can have a higher time-decayed weight than a more persistent item*. e.g., for the items s_1 and s_2 as defined above, we take the weights of the items when they appeared be 1, and compute the time-decayed weights under the decay function $f(c - t_i) = e^{-(c-t_i)}$ at $c = 10$. In this case, the time-decayed weight of s_1 is $f(9)+f(8)+f(7)+f(6)+f(5) = 0.01058$, whereas that for s_2 is $f(9)+f(7)+f(5)+f(3)+f(1) = 0.425$. Although s_1 was twice as persistent as s_2 , it appeared long ago and then never re-appeared; whereas s_2 appeared at some evenly spread-out timeslots over the entire window, and thus managed to have a higher time-decayed weight than s_1 . Comparing the recency and the persistence of stream items may be an interesting research topic in itself.

Kleinberg [35] also studied the temporal behavior of data stream items. However, his work focused on developing a formal approach for modeling “bursts” in data streams, i.e., items which have a sudden rise in their frequencies over certain intervals of time. We want to formalize the notion of persistence in general, so that we can compare the persistence of two data items in a stream, none of which is necessarily “bursty”.

2.2 Problem Formulation

We consider a stream $S = (a_1, \dots, a_m)$ of data items, where each $a_i \in [n] = \{1, \dots, n\}$, and the indices denote the timestamps, so that we have one arrival per timeslot. For any $v \in [n]$, which occurs twice or more in the stream, let $t_1^v, t_2^v, t_3^v, \dots, t_k^v$ be the subset of timestamps when v was observed. Then, we define the persistence of v as $p_v = \sum_{j=1}^{k-1} \frac{1}{t_{v_{j+1}} - t_{v_j}}$. We define the following two problems:

Problem 1 : Identification of persistent sources : *For some user-input $\phi \in (0, 1)$, identify all items v which are ϕ -persistent, i.e., for which $p_v > \phi(m - 1)$.*

Problem 2 : Identification of stealthy sources : *For some user-input $0 < \phi < \psi < 1$, identify all items v for which $\phi(m - 1) < p_v < \psi(m - 1)$.*

Note that, in Problem 2, the lower threshold ensures that the data item v appears pretty frequently, and the upper threshold ensures that it does not appear at timeslots that are “too close” to each other.

Chapter 3

Statistics of Frequent Items

3.1 Motivation

Xu *et al.* [48] studied the behavior profile of Internet backbone traffic in terms of communication patterns of end-hosts and services. The flow data they worked on was 4-dimensional: (source IP address, destination IP address, source port, destination port). They extracted the heavy-hitters along each dimension. Their analysis, based on a 24-hour packet trace (from which heavy-hitters were extracted once every five minutes along each of the four dimensions), revealed the following patterns:

- While the total number of distinct values along a given dimension may not fluctuate very much, the number of heavy-hitters may vary dramatically, due to changes in the underlying feature value distributions.
- The number of heavy-hitters is far smaller than the distinct number of feature values. In other words, the distribution of feature values is highly *skewed*.
- They *dynamically* found out the cut-off threshold $\hat{\alpha}^*$ for identifying the heavy-hitters along each dimension. They observed that $\hat{\alpha}^*$ for the different dimensions also differ.
- No single *fixed* threshold was adequate for defining a feature value as a heavy-hitter.

This shows that the study of heavy-hitters is always relevant for any research with network data. In fact, in the data stream literature, there already is extensive work on identifying the heavy-hitters [11, 16, 38, 39] in

a stream in small space. We have seen in Section 1.2 how this problem can be formally defined. There are many existing algorithms that identify the frequent elements from a stream, and provide estimates of their frequencies too, but do not yield any further information about the frequent items. For example, in a stream of (source IP, destination IP) tuples, in addition to knowing the frequently occurring source IP or destination IP addresses, we may be interested in the following aggregates:

- **Identifying frequent destinations for frequent sources:** Consider the following simplistic scenario of a DoS attack. Suppose there are four attackers: s_1, s_2, s_3 and s_4 , and four targets: d_1, d_2, d_3 and d_4 . Suppose we have a central hub/router and all traffic among the attackers and the targets are routed through the hub. Suppose we collect 1,000 (source IP, destination IP) tuples over a short time interval, in which 350 tuples have s_1 as the source IP, and among these 350 tuples, 75 have d_1 as the destination IP. Suppose, s_1 is conducting a DoS attack on d_1 alone, and no other attacker is sending any traffic to d_1 . The most straightforward way to identify the attackers (victims) in a DoS attack is to identify the frequent source (destination) IP addresses. But then, if we use a single threshold $\phi = 0.3$ in this case to identify the attackers as well as the victims, then s_1 would be identified as an attacker, but d_1 would not be identified as a target. This calls for scrutinizing the traffic from the frequent source s_1 more closely to see what its frequent destinations are, and justifies the need of a separate threshold. If a threshold $\psi = 0.2$ is applied on the outbound traffic from s_1 , then d_1 can be identified as a potential victim of the attack.
- **Computing the frequency moments on destination IP addresses for the frequent source IP addresses:** We have already presented the definition of the k^{th} frequency moment from [6] in Section 1.2. Note that, for each of the frequently occurring source IP addresses in a traffic stream, we have a frequency distribution of destination IP addresses that occur alongwith that source IP address. So, it is worth attempting to compute the frequency moments on the destination IP addresses for the frequent source IP addresses. Once we know a source IP address is frequent, we would be naturally interested to know the number of distinct destination IP addresses this source sends traffic to (F_0), or a measure of the skew (F_2) of the distribution of the destination IP addresses this source sends traffic to.

- **Identifying frequent sources for frequent destinations:** This is reverse of the first of the previous two problems. Hence, we can construct a similar scenario, with the frequencies for sources and destinations reversed, to justify the significance of this problem.

3.2 Related Work

In the data streaming literature, there is a significant body of work on the design of sketches for identification of heavy hitters [11, 16, 38, 39]. A fairly obvious approach would be to extend the Misra-Gries algorithm [39] to maintain a separate sketch for each frequently occurring source IP address, so that we can identify the frequently occurring destination IP addresses for each frequent source IP address. The problem is, our general understanding of data streams reveals that it is not possible to come up with a *single* (small-space) sketch that can compute the frequency moments approximately, and at the same time can identify the heavy-hitters (approximately). This is because any approximate algorithm for identifying the heavy-hitters cuts down on the space requirement by discarding the non-frequent items from the sketch. So, if we want to estimate the frequency moments (particularly F_0) based on such a sketch, then with high probability, our estimates will be far from accurate, because the non-frequent items can contribute a lot to F_0 .

In the context of data stream algorithms, Cormode *et al* [12, 14], worked on the problem of identifying hierarchical heavy-hitters (HHH). It arises when the data comes from a domain which is itself hierarchical, e.g., IP addresses, in dotted decimal notations, can be considered to come from a hierarchical domain of height 4, since there are 4 octets in the address. In this case, 135.207.*.* is a parent of 135.207.50.* in the hierarchy, since the leading two octets are same for both. Alternatively, when represented in 32 bits, IP addresses can be considered to come from a hierarchical domain of height 32. In this case, 135.207.50.250/24 is a parent of 135.207.50.250/25 in the hierarchy, since the leading 24 bits of the two addresses are the same. We now present the model and the formal definition of HHHs from [12]: we are given a (multi)set S of elements from a hierarchical domain D of height h . Let $elements(T)$ be the union of elements that are descendants of a set of prefixes T of the domain hierarchy, and f_e denote the frequency of each element e in S . Given a threshold ϕ , the set of HHHs of S at level i is defined in the following inductive way:

Definition 3.2.1 $HHH_0 = \{e : f_e \geq \lfloor \phi N \rfloor\}$. For $i \geq 1$, given a prefix p at level i in the hierarchy,

the frequency of p is defined as $F(p) = \sum f(e) : e \in \text{elements}(\{p\}) \wedge e \notin \text{elements}(\cup_{l=0}^{i-1} \text{HHH}_l)$, i.e., the sum of frequencies of all the elements that are descendants of p in the domain hierarchy, but are not themselves heavy-hitters upto level $i - 1$. Then, the set of HHHs at level i in the hierarchy is $\text{HHH}_i = \{p : F(p) \geq \lfloor \phi N \rfloor\}$. The set of all hierarchical heavy-hitters of S is given by $\text{HHH} = \cup_{i=0}^h \text{HHH}_i$.

In a follow-up paper [13], Cormode *et al* extended the HHH problem to two and higher dimensional data streams, with several alternative definitions. We do not present all their definitions formally, but, given the informal definitions of our problems in Section 3.1, and the definition of HHHs that we just presented, we are now in a position to explain why our problem is different from identifying HHHs. Consider a stream of (srcIP, dstIP) tuples. If the tuple (a, b) (a and b are IP addresses) is a hierarchical heavy hitter, then so is any tuple of the form $(*, b)$ or $(a, *)$, where $*$ stands for any IP address, because from the definition of hierarchical heavy hitters, it follows that the frequency of the permutation (a, b) in the stream exceeds a threshold ϕ , which implies the frequency of any tuple of the form $(*, b)$ or $(a, *)$ exceeds the threshold ϕ too. But we want to identify all source IP addresses a such the total (relative) frequency of the tuples of the form $(a, *)$ exceeds a threshold ϕ , and then, from the tuples of the form $(a, *)$, we want to identify all IP addresses b such that the (relative) frequency of the tuple (a, b) , among the tuples of the form $(a, *)$, exceeds ψ (ψ and ϕ may be same or different). Note that b occurs frequently with a does not necessarily imply that b would be frequently occurring with other source IP addresses also. In fact, b might occur frequently with some source IP address a , where a may not even qualify as a frequent source IP address.

In the context of real NIDS development, Gonzalez and Paxson [26] brought in some improvements in the Bro NIDS to identify the large connections (or flows), using the fact that the amount in traffic in a stateful TCP connection can be easily calculated by computing the difference between the sequence numbers at the beginning and at the end of a connection. They also identified the heavy-hitters at several granularity levels, e.g., identifying the busiest connections, the busiest host pairs, the busiest sources and destinations, etc. Their heuristics were based on the random sampling technique, and the observation that given unbiased sampling, a heavy-hitter in the full traffic stream is very likely to be a heavy-hitter in a sampled traffic stream too. However, they do not present any theoretical guarantees on the performances of their techniques.

3.3 Problem Formulation

We formally define, on 2-dimensional streams, a set of problems, all of which provide extensive statistics about the distributions of one attribute, given some value of the other attribute is a heavy-hitter in the stream. Our model is: we are given a data stream S on network traffic flows, where each data element is a tuple of the form (s_i, d_i) , where s and d are the source and destination IP addresses in the IP header of a packet. Here are the problems:

Problem 3 : Identifying frequent destinations for frequent sources: *For a stream of size N , and user-defined thresholds ϕ and ψ , our goals are the following:*

1. *To identify any source IP address s such that*

$$f_s = |\{(s_i, d_i) : (s_i, d_i) \in S \text{ and } s_i = s\}| > \phi N$$

2. *For any source IP address s as above, to identify any destination IP address d such that*

$$f_{s,d} = |\{(s_i, d_i) : (s_i, d_i) \in S \text{ and } s_i = s \text{ and } d_i = d\}| > \psi f_s$$

i.e., if a source IP address s has f_s occurrences in S , then we want to identify any destination IP address that occurs more than ψf_s times along with s .

The problem of identifying frequent sources for frequent destinations can be done in a manner similar to Problem 3.

Problem 4 : Computing frequency moments for destinations for frequent sources: *For a stream of size N , we define the k^{th} frequency moment of any source IP address s as $F_s^{(k)} = \sum_d f_{s,d}^k$, where $f_{s,d} = |\{(s_i, d_i) : (s_i, d_i) \in S \text{ and } d_i = d \text{ and } s_i = s\}|$. For user-defined inputs ϕ and k , our goals are the following:*

1. *To identify any source IP address s such that*

$$f_s = |\{(s_i, d_i) : (s_i, d_i) \in S \text{ and } s_i = s\}| > \phi N$$

2. To compute $F_s^{(k)}$ for any source IP address s as above.

Note that for Problem 4, we would be mostly interested in $F_s^{(k)}$ for $k = 0$ and $k = 2$. We might have to come up with separate sketches for $F_s^{(0)}$ and $F_s^{(2)}$.

Rather than treating all the tuples with equal weight, we can pay more attention to the more recent ones, i.e., we can introduce a time-decay model like [17] for this problem, as discussed in Section 2.1. For that, let S be a stream of 3-tuples of size N , i.e., tuples of the form (s_i, d_i, t_i) where t_i is the timestamp. We have a function $g(\cdot)$ that is monotonically decreasing. We define the time-decayed size of the stream, the time-decayed frequency of a source IP address s and the time-decayed frequency of a (source IP, destination IP) pair (s, d) in the following manner :

Definition 3.3.1 *The time-decayed size of the stream S at the current time c is*

$$N^{(c)} = \sum_{(s_i, d_i, t_i): (s_i, d_i, t_i) \in S} g(c - t_i)$$

Definition 3.3.2 *The time-decayed frequency of a source IP address s , at the current time c is*

$$f_s^{(c)} = \sum_{(s_i, d_i, t_i): (s_i, d_i, t_i) \in S \text{ and } s_i = s} g(c - t_i)$$

Definition 3.3.3 *The time-decayed frequency of a (source IP, destination IP) pair (s, d) , at the current time c is*

$$f_{s,d}^{(c)} = \sum_{(s_i, d_i, t_i): (s_i, d_i, t_i) \in S \text{ and } s_i = s \text{ and } d_i = d} g(c - t_i)$$

We are now in a position to define the time-decayed version of Problem 3:

Problem 5 : Identifying recent frequent destinations for recent frequent sources: *For a stream of size N , and user-defined thresholds ϕ and ψ , our goals at current time c are the following:*

1. To identify any source IP address s such that

$$f_s^{(c)} > \phi N^{(c)}$$

2. For any source IP address s as above, to identify any destination IP address d such that

$$f_{s,d}^{(c)} > \psi f_s^{(c)}$$

Note that, if we can solve this generalized time-decayed version of the problem, that would lead to a solution for the sliding-window model, since the sliding-window model (for a time-window of size W) is a special case of the time-decayed problem where $g(\cdot)$ is defined as

$$g(x) = \begin{cases} 1, & \text{if } x \leq W \\ 0, & \text{otherwise} \end{cases} \quad (3.1)$$

3.4 Progress: An Approximation Solution

We present the summary of some results we have already got while attempting Problem 3. We adopted the Misra-Gries algorithm [39] for finding frequent elements. Since the single-pass Misra-Gries algorithm cannot solve the problem exactly, but comes up with an ϵ -approximate solution, we keep two approximation parameters, ϵ for the identification of the frequent source IP addresses, and δ for the identification of the frequent destination IP addresses corresponding to the frequent source IPs. So, our algorithm would provide the following guarantees:

1. Any source IP address s such that $f_s > \phi N$ will be identified.
2. No source IP address s such that $f_s < (\phi - \epsilon)N$, will be reported.
3. For any source IP address s as above, any destination IP address d such that $f_{s,d} > \psi f_s$ will be identified.
4. For any source IP address s as above, no destination IP address d such that $f_{s,d} < (\psi - \delta)f_s - \delta'N$, where $\delta' = (\psi - \delta + 1)\epsilon$, will be reported.

The Misra-Gries algorithm, at any point of time, maintains at most $O(\frac{1}{\epsilon})$ counters. In our algorithm, we maintain at most $O(\frac{1}{\epsilon'})$ counters for each of the $O(\frac{1}{\epsilon})$ source IP addresses in H . So the size of our sketch is at most $O(\frac{1}{\epsilon\epsilon'}) = O(\frac{1}{\epsilon\delta})$.

Chapter 4

Identifying Heavy-Hitters from Distributed Datasets

4.1 Motivation

As we have already discussed in Section 1.3, in a DDoS attack, the intruders typically capture some masters and many daemons to simultaneously send excessive traffic towards a single victim (typically a web server), so that legitimate clients are denied service. Detecting a DDoS attack is equivalent to finding that the total number of accesses to some server has exceeded a threshold. A distributed frequent elements algorithm can help by tracking the most frequently accessed web servers in a distributed manner, and noting if these frequencies are abnormally large. We devised a *gossip-based* algorithm [36], where the nodes (masters and daemons) select communication partners uniformly at random, and exchange data about their own access patterns. Thus the identification of the frequently accessed web servers can proceed in a totally decentralized manner.

4.2 Related Work

Demers *et al.* [19] were the first to provide a formal treatment of gossip protocols (or “epidemic algorithms” as they called them) for data dissemination. Kempe and Kleinberg [33] analyzed the influence of the underlying gossip mechanism on the design of gossip-based protocols, and explored the limitations of

uniform gossip in solving the *nearest resource location problem*. Kempe, Dobra and Gehrke [32] proposed algorithms for computing the sum, average, approximately uniform random sampling and quantiles using uniform gossip. Their algorithm for quantiles are based on their algorithm for the sum – they choose a random element and count the number of elements that are greater and lesser than the chosen element, and recurse on smaller data sets until the median is found. Thus their algorithms need many instances of “sum” computations to converge before the median is found. A similar approach could potentially be used to find frequent elements using gossip. In contrast, our algorithms for frequent elements are not based on repeated computation of the sum, and converge faster.

Much recent work [7, 8, 41] has focused on computing “separable functions” using gossip. A separable function is one that can be expressed as the sum of individual functions of the node inputs. For example, the function “count” is separable, and so is the function “sum”. However, the set of frequent elements is not a separable function. Hence, these techniques do not apply to our problem. There is much other work on the computation of basic aggregates, we list a few representative ones here. Kashyap *et al.* [31] proposed algorithms for gossip with flexible tradeoffs between the number of rounds and the number of messages transmitted. Dimakis, Sarwate and Wainwright [20] consider the problem of computing the average over *random geometric graphs* with location-aware nodes, combining uniform gossip with greedy geographic routing.

The problem of identifying frequent elements in data has been extensively studied [39, 38, 30] in the database, data streams and network monitoring communities (where frequent elements are often called “heavy-hitters”). The early work in this is due to Misra and Gries [39], who proposed a deterministic algorithm to identify frequent elements in a stream, followed by Manku and Motwani [38], who gave randomized and deterministic algorithms for tracking frequent elements in limited space. The above were algorithms for a centralized setting.

Cao and Wang [10] proposed an algorithm to find the top- k elements in a distributed setting, where they first made a lower-bound estimate for the k^{th} value, and then used the estimate as a threshold to

prune away elements which should not qualify as top- k . Zhao *et al.* [51] proposed a sampling-based and a counting-sketch-based scheme to identify globally frequent elements. Manjhi *et al.* [37] present an algorithm for finding frequent items on distributed streams, through a tree-based aggregation. Venkataraman *et al.* [47] present an algorithm for identifying “superspreaders” or “heavy distinct hitters” in a network data stream. Keralapura, Cormode and Ramamirtham [34] proposed an algorithm for continuously maintaining the frequent elements over a network of nodes. The above algorithms are not directly applicable to the problem of identifying frequent elements using gossip, because they sometimes assume the presence of a central node, or an underlying network structure such as a spanning tree [37, 34], and further, they do not consider the dissemination of sketch through gossip, or the cost of network-wide communication

4.3 Problem Formulation

In [36], we considered a distributed system with N nodes numbered from 1 to N . Each node i holds a single data item m_i . Without loss of generality, we assume that $m_i \in \{1, 2, \dots, m\}$ is an integer representing an item identifier. For data item $v \in \{1, \dots, m\}$, the frequency of v is denoted by f_v , and is defined as the number of nodes that have data item v , i.e. $f_v = |\{j \in [N] : m_j = v\}|$. Note that f_v may not be available locally at any node, in fact determining f_v itself requires a distributed computation. The task is to identify those elements that have large frequencies. We consider the scenario of *uniform gossip*. We present the *asynchronous* temporal model here, where time is divided into non-overlapping rounds. In each round, a single source node, chosen uniformly at random out of all N nodes, transmits to another randomly chosen receiver. The time complexity of the algorithm is the number of rounds the gossip has to be continued for to accomplish the goal. We considered the following two variants of the problem, depending on how the thresholds are defined.

Absolute Threshold. The user gives an absolute frequency threshold $k > 1$ and approximation error λ ($\lambda < k$). An item v is considered a frequent item if $f_v \geq k$, and v is an infrequent item if $f_v < k - \lambda$. Note that with a data set of N elements there may be up to N/k frequent elements according to this definition.

Relative Threshold. In some cases, the user may not be interested in an absolute frequency threshold,

but may only be interested in identifying items whose relative frequency exceeds a given threshold. More precisely, given a relative threshold ϕ ($0 < \phi < 1$), approximation error ψ ($0 < \psi < \phi$), an item v is considered to be a frequent item if $f_v \geq \phi N$, and v is considered an infrequent item if $f_v < (\phi - \psi)N$. According to this definition, there may be no more than $1/\phi$ frequent items.

4.4 Research Results

4.4.1 Absolute Threshold

Our algorithm is based on random sampling. The elements of S are sampled in a distributed manner, and the sampled elements are disseminated to all nodes using gossip – the cost of doing so is small, since the random sample is typically much smaller than the size of the population. The sampling also ensures that frequent elements are exchanged more often during the later gossip phase. Intuitively, suppose we sample each element from S into a set T with probability $1/k$. For a frequent element v with $f_v \geq k$, we (roughly) expect one or more copies of v to be included within T . Similarly, for an infrequent element u with $f_u < k - \lambda$, we expect that no copy of u will be included in T . However, some infrequent elements may get “lucky” and may be included in T and similarly, some frequent elements may not make it to T . The probabilities of these events decrease as the sample size increases.

To refine this sampling scheme, we sample with a probability that is a little larger than $1/k$, say c/k for some parameter c . Finally, we select those elements that occur at least r times within T , for some parameter $r < c$ that will be decided by the analysis. It turns out that c and r will need to depend on the approximation error λ as well as the threshold λ . The smaller λ is, the greater should be the sampling probability, since we need to make a more precise distinction between the frequencies of frequent and infrequent elements. In the actual algorithm, we use a sampling probability of $\frac{12k}{\lambda^2} \ln \frac{2}{\delta}$ – note that this is $\Omega(\frac{1}{k})$ since $\lambda < k$ and hence $\frac{k}{\lambda^2} > \frac{1}{k}$.

We analytically established the following claims for the absolute threshold version of the problem.

Lemma 4.4.1 False Negative. *If v is an element with $f_v \geq k$, then with probability at least $1 - \delta$, v is*

returned as a frequent element by every node after $20N \ln N$ rounds of gossip.

Lemma 4.4.2 False Positive. *If u is an element with $f_u \leq k - \lambda$, where $k^{\frac{3}{4}} \leq \lambda < k$, then the probability that u is returned by some node as a frequent element is no more than δ .*

Let \mathcal{Y} denote the total number of bytes that need to be exchanged for the whole protocol until the frequent elements have been identified. We got the following result about the communication complexity of the algorithm for absolute threshold.

Theorem 4.4.1 (Communication Complexity for Absolute Threshold) *With high probability,*

$$\mathcal{Y} = O\left(\frac{N^2 k}{\lambda^2} \ln \frac{1}{\delta} \ln N\right)$$

4.4.2 Relative Threshold

For the relative threshold variant of the problem, unlike the case of absolute threshold, there is no fixed probability that a node can use to sample data elements locally. For the same relative frequency threshold, the absolute frequency threshold (ϕN), as well as the approximation error (ψN) increases with N . Thus if ϕ and ψ are kept constant and N increases, then a smaller sampling probability will suffice. Since we do not have prior knowledge of N , we need a more “adaptive” method of sampling where the sampling probability decreases as more elements are encountered during gossip.

To design our sketch, we used an idea similar to *min-wise independent permutations* [9]. Each data item $m_i, i = 1 \dots N$ is assigned a weight w_i , which is a random number in the unit interval $(0, 1)$. The algorithm maintains a sketch T of (m_i, w_i) tuples that have the t smallest weights w_i . The value of t can be decided independent of the population size N . The intuition is that if an element v has a large relative frequency, then v must occur among the tuples with the smallest weight. Maintaining these minimum-weight elements through gossip is easy, and if we choose a large enough sketch, the likelihood of a frequent element appearing in the sketch a sufficient number of times is very high. We identify a value m as a frequent item if there are at least $(\phi - \frac{\psi}{2})t$ tuples in T with $m_i = m$; otherwise, m is not identified as a frequent element. The threshold t is chosen to be $O(\frac{1}{\psi^2} \ln(\frac{1}{\delta}))$.

The following lemmas provide upper bounds on the probabilities of finding a false negative and a false positive respectively, by our algorithm.

Lemma 4.4.3 *If the algorithm is run for $20N \ln N$ rounds, then, with probability at least $1 - \delta$, v is identified by every node as a frequent element, if v is a frequent element, i.e. $f_v \geq \phi N$.*

Lemma 4.4.4 *If the algorithm is run for $20N \ln N$ rounds, then, with probability at least $1 - \delta$, u is not identified by any node as a frequent element, if u is an infrequent element, i.e. $f_u < (\phi - \psi)N$.*

Since the size of the sketch at any time during gossip is at most $t = O(\frac{1}{\psi^2} \ln(\frac{1}{\delta}))$, we get the following result on the communication complexity.

Theorem 4.4.2 *The number of bytes exchanged by the algorithm till the frequent elements are identified is at most $O(\frac{1}{\psi^2} \ln(\frac{1}{\delta})N \ln N)$, with probability at least $1 - O(\frac{1}{N})$.*

Chapter 5

Conclusion

In this research, we attempt to formalize the notion of exploit patterns, and, design and analyze efficient algorithms for detecting these patterns from network flow streams. We discussed the current state of the art in signature-based network intrusion detection systems, and pointed out the areas where data streaming algorithms, with *provable* performance guarantees, can be applied to detect intrusion. We presented an overview of the work we have done so far and a plan of our future work.

Bibliography

- [1] <http://www.tcpdump.org/>.
- [2] <http://www.wireshark.org/>.
- [3] <http://aircert.sourceforge.net/yaf/>.
- [4] <http://qosient.com/argus/>.
- [5] <http://oss.oetiker.ch/mrtg/>.
- [6] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.
- [7] Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip algorithms: design, analysis and applications. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, pages 1653–1664, 2005.
- [8] Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.
- [9] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing (STOC)*, pages 327–336, 1998.
- [10] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 206–215, 2004.
- [11] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.
- [12] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 464–475, 2003.

- [13] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 155–166, 2004.
- [14] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in streaming data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(4), 2008.
- [15] Graham Cormode and S. Muthukrishnan. What’s hot and what’s not: tracking most frequent items dynamically. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data / Principles of Database Systems (PODS)*, pages 296–306, 2003.
- [16] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
- [17] Graham Cormode, Srikanta Tirthapura, and Bojian Xu. Time-decaying sketches for sensor data aggregation. In *Proceedings of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 215–224, 2007.
- [18] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal of Computing*, 31(6):1794–1813, 2002.
- [19] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Principles of Distributed Computing (PODC)*, pages 1–12, 1987.
- [20] Alexandros G. Dimakis, Anand D. Sarwate, and Martin J. Wainwright. Geographic gossip: efficient aggregation for sensor networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN)*, pages 69–76, 2006.
- [21] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *USENIX*, 2006.
- [22] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.
- [23] John Gerth. Incorporating network flows in intrusion incident handling and analysis. In *FLOCON*, 2008.
- [24] Phillip B. Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, 2001.

- [25] Phillip B. Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 63–72, 2002.
- [26] José M. González and Vern Paxson. Enhancing network intrusion detection with integrated sampling and filtering. In *Proceedings of the 9th International Symposium On Recent Advances In Intrusion Detection (RAID)*, pages 272–289, 2006.
- [27] Paul Helman, Gunar E. Liepins, and Wynette Richards. Foundations of intrusion detection. In *The 5th IEEE Computer Security Foundations Workshop (CSFW)*, pages 114–120, 1992.
- [28] L. Todd Herberlein, Gihan V. Dias, Karl N. Levitt, Biswanath Mukherjee, Jeff Wood, and David Wolber. A network security monitor. In *IEEE Symposium on Security and Privacy*, pages 296–305, 1990.
- [29] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, pages 211–225, 2004.
- [30] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.
- [31] Srinivas R. Kashyap, Supratim Deb, K. V. M. Naidu, Rajeev Rastogi, and Anand Srinivasan. Efficient gossip-based aggregate computation. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 308–317, 2006.
- [32] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS)*, pages 482–491, 2003.
- [33] David Kempe and Jon M. Kleinberg. Protocols and impossibility results for gossip-based communication mechanisms. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS)*, pages 471–480, 2002.
- [34] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 289–300, 2006.
- [35] Jon M. Kleinberg. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery*, 7(4):373–397, 2003.
- [36] Bibudh Lahiri and Srikanta Tirthapura. Computing frequent elements using gossip. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 119–130, 2008.

- [37] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (recently) frequent items in distributed data streams. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 767–778, 2005.
- [38] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 346–357, 2002.
- [39] Jayadev Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.
- [40] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems*, 24(2):115–139, 2006.
- [41] Damon Mosk-Aoyama and Devavrat Shah. Computing separable functions via gossip. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 113–122, 2006.
- [42] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.
- [43] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.
- [44] David Plonka. Flowscan: A network traffic flow reporting and visualization tool. In *Proceedings of the 13th Systems Administration Conference (LISA)*, pages 305–317, 2000.
- [45] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference (LISA)*, pages 229–238, 2000.
- [46] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1/2):105–136, 2002.
- [47] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B. Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.
- [48] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. In *Proceedings of the Annual Conference of the Special Interest Group on Data Communication (SIGCOMM)*, pages 169–180, 2005.
- [49] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Reducing unwanted traffic in a backbone network. Appeared in the Proceedings of the Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI), 2005.

- [50] Vinod Yegneswaran, Paul Barford, and Johannes Ullrich. Internet intrusions: global characteristics and prevalence. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 138–147, 2003.
- [51] Qi Zhao, Mitsunori Ogihara, Haixun Wang, and Jun Xu. Finding global icebergs over distributed data sets. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 298–307, 2006.