Entropy Based Adaptive Flow Aggregation

Yan Hu, Dah-Ming Chiu, *Fellow, IEEE*, and John C. S. Lui, *Senior Member, IEEE*

***Abstract*—Internet traffic flow measurement is vitally important for network management, accounting and performance studies. Cisco's NetFlow is a widely deployed flow measurement solution that uses a configurable static sampling rate to control processor and memory usage on the router and the amount of reporting flow records generated. But during flooding attacks the memory and network bandwidth consumed by flow records can increase beyond what is available. Currently available countermeasures have their own problems: (1) reject new flows when the cache is full - some legitimate new flows will not be counted; (2) export not-terminated flows to make room for new ones - this will exhaust the export bandwidth; (3) adapt the sampling rate to traffic rate - this will reduce the overall accuracy of accounting, including legitimate flows. In this paper, we propose an entropy based adaptive flow aggregation algorithm. Relying on information-theoretic techniques, the algorithm efficiently identifies the clusters of attack flows in real time and aggregates those large number of short attack flows into a few metaflows. Compared to currently available solutions, our solution not only alleviates the problem in memory and export bandwidth, but also significantly improves the accuracy of legitimate flows. Finally, we evaluate our system using both synthetic trace file and real trace files from the Internet.**

***Index Terms*—Traffic measurement, Network monitoring, Data summarization, Information theory.**

I. Introduction

TRAFFIC measurement and monitoring are crucial to operating IP networks, because network administrators need to have a good understanding of how their networks are used and misused. Some existing systems operating on relative low traffic links can perform complex security analysis to reveal malicious activities [1], [2], or simply capture packet (header) traces to be analyzed offline. SNMP counters [3] are a simpler solution more widely deployed on the high-speed links, but they only report the total amount of traffic transmitted on the measured link. Flow-level measurement, such as done in the Cisco's NetFlow [4], offers a good compromise between scalability and complexity since it can offer detail information about the traffic crossing the network.

The ever increasing speeds of transmission links and high volume of traffic present great challenges for flow measurement. The first challenge is at the point of measurement. For high-speed interfaces, the processor and the flow memory of the router cannot keep up with the high packet rate. Another problem is that the volume of complete measurements of all traffic requires too many resources, both in the bandwidth required to transmit the flow records to the collector, and the resources needed to store and process the records at the collector.

These scalability issues motivate using some form of data reduction. A standard solution is to perform packet sampling. In Cisco's sampled NetFlow, the router forms flow reports from a sampled substream of all packets passing through it. The problem is that the sampling rate of Cisco NetFlow is usually set manually by network operators according to the normal traffic volume in their network. When there is an anomaly in the network, such as DoS attacks, worm spread, aggressive port scans and flash crowds, which generates a large number of small flows, the surge in the number of small flows may overwhelm the router memory and the export bandwidth to the collector.

Current countermeasures to the above problem include: 1) Reject new flows when the cache is full. In this case, legitimate new flows will not be accounted for and the operator will lose the flow data; 2) When the cache is full, export the flow records more aggressively for those non-terminated flows so as to make room for new ones. The implication of this action is that the export bandwidth demand will be very high and may run into trouble at the collector or the way to the collector; 3) Estan et al. in [5] propose a method of adapting the sampling rate to traffic. This algorithm guarantees a stable flow cache and export bandwidth even under severe DoS attacks. But under DoS attacks the sampling rate will decrease to a very low level, which results in poor overall accuracy in per flow counting including legitimate flows.

Our solution is to implement *adaptive flow aggregation* when the router is running low on memory resource. Note that attacks usually have some common patterns: DoS attacks often have the same destination IP address, while worm spreads have the same source IP address. If we dynamically aggregate the large number of such small flows into a few flows, then we can alleviate the problem of memory shortage under attacks. Compared to other countermeasures, our method has several advantages:

- We do not need to decrease the sampling rate drastically under attacks, neither would we reject new legitimate flows because the cache is full. So we significantly improve the accuracy of legitimate flows.
- Without aggressively exporting the records of non-terminated flows so as to make room for new ones, we avoid overwhelming the collector.
- Using the flow aggregation results, we can provide network administrators some useful information to detect DoS attacks and worm spreads.

Therefore, the objective of our system is to identify and aggregate the abnormal flows while keeping legitimate flows unaffected when the router is running low on memory under

Manuscript received June 9, 2006; revised April 6, 2007, and December 13, 2007; approved by IEEE/ACM Transactions on Networking Editor Matthew Roughan.

Yan Hu and Dah-Ming Chiu are with the Department of Information Engineering, the Chinese University of Hong Kong (email: yhu4@ie.cuhk.edu.hk, dmchiu@ie.cuhk.edu.hk).

John C. S. Lui is with the Department of Computer Science & Engineering, the Chinese University of Hong Kong (email: cslui@cse.cuhk.edu.hk).

abnormal conditions. In more detail, it can be stated as: 1) Identify traffic clusters that contain packets of abnormal traffic, and retain as many key attributes as possible when merging the flows in these clusters to metaflows. 2) Pick out the normal flows mixed with abnormal traffic in the identified clusters. 3) Obtain as high accuracy as possible when estimating flow statistics of any aggregates of the traffic.

In this paper, we propose an entropy based adaptive flow aggregation algorithm, which meets these requirements satisfactorily. Based on the concept of entropy from information theory, we keep track of different traffic clusters and use an index, APP (Aggregation Priority Parameter), to indicate each cluster's priority for aggregation. An efficient algorithm is used to identify those clusters as well as pick out some large normal flows belonging to the identified clusters.

The rest of the paper is organized as follows. We describe background and related work in Section II. In Section III, we provide the definition of the *cluster* as well as the properties of those clusters that we choose to do flow aggregation. We describe the data structure we use in Section IV. After that, we present the entropy based flow aggregation algorithm in Section V and provide some analysis in Section VI. Experimental evaluation based on the proposed method is presented in Section VII. Conclusion is given in Section VIII.

II. **Background and Related work**

A. **NetFlow**

NetFlow [4], first implemented in Cisco routers, is the most widely used flow measurement solution today. Flows are defined by seven keys: source and destination IP address, protocol, source and destination port, type of service and input interface. Routers running NetFlow maintain a "flow cache" to keep active flows passing through it. When a packet arrives at the router, the router determines if this packet belongs to an active flow in the cache. If yes, relevant fields (number of packets, number of bytes, timestamp of the last packet, etc) of this flow are updated. If not, the router inserts a new flow record into the flow cache. The router will terminate a flow in its cache if any one of these criteria are met: 1) the interpacket time within the flow exceeds the *inactive timer* (15 sec is the default); 2) this flow record had creation time before the current *active timer* (30 min is the default); 3) observation of TCP flags (FIN or RST); 4) the flow cache is full. For those terminated flows, their records will be exported using UDP to collectors for future analysis.

For high-speed interfaces, Cisco introduced sampled NetFlow [6]. To the problem of NetFlow generating too much data, Cisco's solution is to implement router-based flow aggregation [7]. Different aggregation schemes summarize NetFlow data on the router before the data is exported to the collector, resulting in lower bandwidth requirement. The IETF working group IPFIX (Internet Protocol Flow Information eXport) also recommends aggregating similar flows into one metaflow [8]. Compared to these predefined aggregation schemes, our goal is to dynamically find flows which form a cluster and aggregate these flows in real time.

B. **Related work**

Recently, a number of studies have investigated flow measurement. Estan et al. in [9] present algorithms that automatically identify large flows. In [10], Choi et al. use adaptive sampling to guarantee that the variance introduced by the variability of packet sizes does not exceed a pre-defined limit. The problem of estimating flow distributions using packet sampling has been studied in [11] and [12]. There are some flow analysis and visualization tools, such as Flowscan ([13]) and CoralReef ([14]).

To deal with traffic surges that exhaust the resources during abnormal situations, Estan et al. propose *adaptive NetFlow* ([5]) which adapts the sampling rate to traffic. They divide the NetFlow operation into measurement bins. They do not terminate flow records during the bin, but terminate all active flow records at the end of the bin. They use a maximum sampling rate at the beginning of each bin, which is determined by the router's CPU capability. During the measurement bin, they dynamically decrease the sampling rate until it is low enough for the flow records to fit into memory.

Traffic characterization and summary have also being studied in a number of works. Estan et al. [15] describe a method of traffic characterization that automatically groups traffic into minimal clusters of conspicuous resource consumption. Instead of using individual flows or other predefined aggregates, they dynamically define multi-dimensional traffic clusters, so that any meaningful aggregate of individual flows is a traffic cluster. The difference with ours is that their objective is to present a good traffic report to the network manager, and their system can be considered as a *post-processing* system instead of a real-time one. In [16], Keys et al. present a system that computes multiple summaries of IP traffic in real time, to produce several kinds of hog (sources or destinations that send or receive many packets, bytes or flows) reports. This system only provides traffic summaries, but does not keep any original flow information as Cisco NetFlow does.

Information-theoretic concepts and approaches have been used to examine a wide variety of networking issues such as traffic matrix estimation [17] and intrusion detection [18]. Xu et al. in [19] use data mining and information-theoretic techniques to build behavior profiles of Internet backbone traffic. In [20], Gu et al. develop a method to detect network anomalies by comparing the current network traffic against a baseline distribution. The baseline distribution is estimated by maximum entropy estimation. Liu et al. in [21] develop an information-theoretic framework to examine the difference in information content when measurements are made at either the flow level or the byte count level, and determine the benefits of compressing traces captured at a single monitoring point.

III. **Cluster**

A. **Defining clusters**

Our mechanism intends to protect NetFlow from overwhelming the memory and the export bandwidth due to rapid increases in traffic from one or more traffic aggregates which we call *clusters*. The first issue we have to address is how many distinct fields are used in constructing traffic clusters?

We choose five fields typically used to define a flow: source IP address, destination IP address, protocol, source port, and destination port. For simplicity, we regard these five fields as four keys: {srcIP, dstIP, srcPort (plus protocol), dstPort (plus protocol)}, because port numbers are meaningful only when combined with protocol type. For example, we use “dstPort = 80,TCP” to represent “dstPort = 80 and protocol = TCP”. Individual flows are defined by unique values for each of these four keys, while clusters are defined by unique values for *some* of these key values. In other words, values for these keys can be a single value, or all possible value (we use * to denote this). For example, a cluster with values {srcIP = *, dstIP = 210.0.0.3, srcPort = *, dstPort = 80,TCP} represents all web traffic to the server with IP address 210.0.0.3.

The justification for choosing these four keys to define clusters is that these four keys are consistent with commonly used keys to define a flow. Additionally, this definition is sufficient for the existing NetFlow data applications such as network planning and application monitoring. Among these four keys, the port numbers and the IP addresses have different sensitivity for the aggregation process. The reason is as follows. First, almost all DoS attacks, worm spread, port scan, and flash crowds have either a common srcIP or dstIP, but not always have a fixed port number. Second, some network applications with a well-known port number such as web traffic with port 80 are always big clusters in the network, but we have no reason to aggregate them to a single flow because they are normal traffic and we aim to maintain more detailed information about these for accounting purposes.

Clusters are flows with the same value in some combinations of these four keys. We illustrate this using some examples. In a Smurf attack [22], the attacker sends a forged ICMP packet to a broadcast address and all receivers respond with a reply to the spoofed IP address (the victim). The cluster for this type of traffic can be represented by ICMP packets to the same dstIP (the victim). In the spreading of the MS-SQL server worm [23], the infected machine will craft and send packets (usually using the same srcPort) to randomly chosen IP addresses on port 1434/UDP. A cluster for this type of worm packets will have the same srcIP (the infected computer) plus the same dstPort (1434/UDP) and the same srcPort. One can find packets of DoS attacks often have a common dstIP (sometimes with a common dstPort); Packets of worm spreads often have the same srcIP (sometimes with a common dstPort); Packets of port scans usually have a common dstIP (sometimes with a common srcIP). Besides these flooding attacks, another network behavior which may cause NetFlow to run out of memory is flash crowds. While its purpose is quite different from DoS attacks, from the network operator’s perspective, these two cases are quite similar. Similar to the DoS attack, a cluster can be defined for packets with the same dstIP (and maybe with the same dstPort).

Based on the above analysis, we regard srcIP and dstIP as more important than the other two keys. So for defining clusters we only consider combinations which at least contain the same srcIP or dstIP. In other words, we would not consider a cluster which only has the same srcPort, and/or the same dstPort. Among the 16 arbitrary combinations of four keys, we would not consider a) clusters with no key, b) clusters with all four keys, and we also ignore the cases like c) clusters that only have srcPort, d) clusters that only have dstPort, and e) clusters that only have srcPort plus dstPort. Finally, we get 11 combinations. These combinations and their corresponding examples are shown in Table I.

combinations	examples
srcIP	most worms
dstIP	smurf attack ([22])
srcIP + dstIP	most portscans
srcIP + srcPort	response from syn flooding victim; response from flash crowds web server
srcIP + dstPort	W32/Blaster worm ([24])
dstIP + srcPort	N/A
dstIP + dstPort	syn flooding attacks ([25]); WWW flash crowds
srcIP + dstIP + srcPort	response from non-IP-spoofing syn flooding
srcIP + dstIP + dstPort	non-IP-spoofing syn flooding attacks
srcIP + srcPort + dstPort	MS-SQL server worm ([23])
dstIP + srcPort + dstPort	DNS flash crowds

TABLE I
The combinations of four keys and some examples.

B. Properties of desired clusters

After describing what constitutes *clusters*, we discuss some properties of the desired clusters, which are related to the objectives of our flow aggregation algorithm. In this section, we discuss these objectives in detail, and thus derive the properties of the clusters that we choose. We use the concept of entropy to express these properties and propose our entropy based flow aggregation algorithm.

1) **Identify clusters containing abnormal traffic:** We intend to protect NetFlow from overrunning resources under abnormal traffic. So the first objective is to identify clusters containing those abnormal traffic. Anomaly detection is an interesting topic for its own sake. We are not trying to construct a system to detect anomalies, but protect NetFlow under anomalies by aggregating flows most likely containing those abnormal traffic. We have described in Section III-A that we focus on clusters of 11 kinds of combinations. Other properties of the clusters that contain abnormal traffic include: first, the number of flows in the clusters is usually large enough to be a problem; second, the size of the flows (the number of packets or bytes) is often much smaller than normal flows; third, some keys other than the fixed value, such as srcIP in DoS attack traffic, dstIP in worm spreading traffic and dstPort in port scan traffic, are often randomly or uniformly distributed. The first objective of flow aggregation is to identify clusters with these properties.

2) **Retain as many key attributes as possible:** When we perform flow aggregation, if we merge all flows in the big cluster with a fixed srcIP/dstIP into one metaflow, we only retain one of the key attributes (the fixed srcIP/dstIP) for these flows, and no longer keep track of the other three key attributes. Merging flows will cause accuracy loss in flow measurement for those key attributes that are not retained. If we can find a smaller and more specific cluster in this big cluster, we can retain more key attributes. For example, in

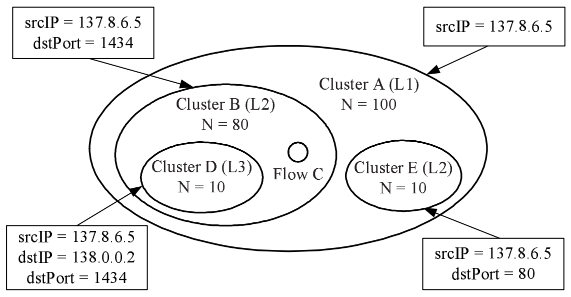


Fig. 1. An example of clusters. N is the number of flows in each cluster.

Fig. 1, cluster A is a big cluster with a fixed srcIP, while cluster B is a smaller and more specific cluster which contains worm spreading traffic from this host. If we choose cluster B (worm spreading traffic) instead of cluster A (total traffic) to do aggregation, we keep track of not only the srcIP but also the dstPort. We define level 1 ($L1$) cluster as the biggest cluster which has either the same srcIP or dstIP, such as cluster A, define level 2 ($L2$) clusters as the clusters which have fixed values in two dimensions, such as cluster B and E, and define level 3 ($L3$) clusters as the clusters which have fixed values in three dimensions such as cluster D. The higher the level of the clusters chosen for aggregation, the smaller the loss in accuracy.

3) **Pick out the larger flows:** If there are several big flows (in terms of bytes or packets) in the identified cluster, we would pick them out. The first reason is that the size of attack flows is often much smaller than that of normal flows, so the big flows in the identified cluster may be normal flows. Secondly, as stated in [26], the omission or inclusion of a bigger flow can have a large effect on estimated total traffic. So we would pick out the big flows from the identified cluster and let them retain all the four key attributes. Because some flows or higher level clusters are picked out, the concept of cluster is extended to the remaining flows in the original cluster. For example, in Fig. 1, large flow C and $L3$ cluster D are picked out from $L2$ cluster B, the remaining flows in cluster B can also be considered as a cluster $F := B - C - D$.

4) **Maintain high accuracy for most aggregates:** Network operators are often uninterested in a single flow, but interested in the aggregates of some flows. For instance, they would like to know how much web traffic is on their link, or which hosts generate the most traffic. These aggregates or clusters often have some significant attributes such as the top applications (port) or hosts (IP). One method to maintain high accuracy for the aggregates that network operators are interested in is to avoid aggregating large flows, as mentioned above. From another point of view, in the clusters that are aggregated, no flow should be significant or stand out from the rest, all flows are nearly indistinguishable.

IV. Data structure

First we take fprobe [27] as an example to illustrate the data structure in ordinary NetFlow process. Fprobe is a libpcap-based tool that collects network traffic data and emits it as NetFlow flow records towards the specified collector. The data structure used to store active flows in this software is a hash table, in which flows are indexed by hash values of their flow ID. The number of flows is often larger than the length of the hash table (in fprobe, there are two choices for the length, 256 and 65536), so two or more flows can be computed to the same hash value. A linked list is used to store flows of this kind of hash collisions. As we have mentioned in Section III-A, we assume flows are defined in terms of four keys, srcIP, dstIP, srcPort, and dstPort. When a packet arrives, the system first computes a hash value H_A on its flow ID (the four keys) using a hash function, $H_A = Hash(srcIP, dstIP, srcPort, dstPort)$. Then the system finds out H_A in the hash table and looks at every flow in the list with H_A, to determine which flow this packet belongs to, or creates a new flow entry if the packet does not belong to any existing flow.

We need a new data structure for our flow aggregation, which is a tradeoff. If we use a simple data structure like a hash table with linked list as mentioned above, it will be inefficient to aggregate flows in a cluster, which needs to traverse every node in the hash table. We need to put flows which are more likely to be aggregated later closer. On the other hand, if we use a complicated data structure like the multi-dimensional tree in [15], it will use excessive memory, and bring too much overhead to normal flow operations like flow look up.

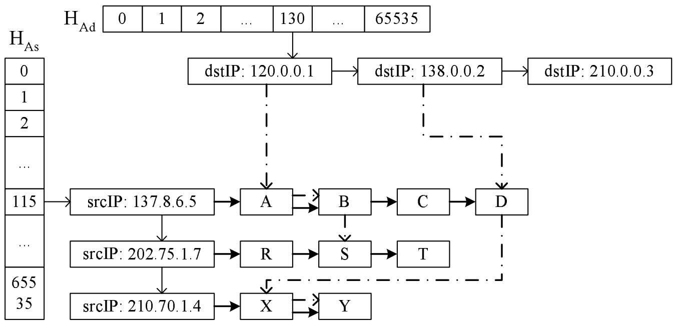


Fig. 2. The data structure for adaptive flow aggregation: a two-dimensional hash table. Node A, B, ... X, Y are flows. Flow A's srcIP is 137.8.6.5 and its dstIP is 120.0.0.1. Flow S's srcIP is 202.75.1.7 and its dstIP is 120.0.0.1.

Our data structure is as shown in Fig. 2. The data structure is a two-dimensional hash table. One dimension of the hash table is H_{As}, the hash value based on a flow's srcIP (the left table of hash number from 0 to 65535). The other dimension is H_{Ad}, the hash value based on a flow's dstIP (the top table of hash number from 0 to 65535). H_{As} of a packet is computed based only on its srcIP, instead of its flow ID of the four keys, that is, $H_{As} = Hash(srcIP)$. Similarly, $H_{Ad} = Hash(dstIP)$.

Take srcIP as an example, packets with the same srcIP will definitely be mapped to the same H_{As}, on the other hand, packets with different srcIPs may be mapped to the same H_{As} because of hash collision. Each H_{As} node has a linked list, which consists of all srcIPs mapped to this H_{As}. For instance, srcIPs of 137.8.6.5, 202.75.1.7 and 210.70.1.4 are all mapped to H_{As} = 115. In addition, every srcIP node has a linked list, which consists of all flows with this srcIP. The dstIP dimension of the hash table has a similar structure. Each H_{Ad} node has a linked list, which consists of all dstIPs mapped to this H_{Ad}.

We only consider clusters containing a fixed srcIP or dstIP, so we compute the hash value based on these two fields. In the srcIP/dstIP list, we put flow ID nodes sorted by dstIP/srcIP. This data structure makes it easier to find flows in one cluster.

In the data structure, every IP node has a counter to indicate the number of flow nodes with this IP address. With this counter, we can easily get a *top list* for the IP addresses with the largest flow numbers. Entries in the top list have a flow counter and a pointer pointing to the corresponding IP address node. Now the problem is that the top list is only for srcIP/dstIP, not for all combinations. The flow aggregation algorithm is to identify the desired clusters from these $L1$ clusters in the top list.

V. Entropy based flow aggregation algorithm

In this section, we describe our entropy based flow aggregation algorithm. We first illustrate a simple flow aggregation algorithm in Section V-A. Then we introduce the concept of entropy to the problem of flow measurement in Section V-B and define a parameter named APP in Section V-C. The detailed description of the entropy based flow aggregation algorithm is given in Section V-D. Finally, we discuss flow aggregation and export in Section V-E.

A. A simple flow aggregation algorithm

Before describing our entropy based flow aggregation algorithm, we first illustrate a simple flow aggregation algorithm, which was proposed and described in [28]. We define m_{max} as the memory usage that triggers aggregation and m_{des} as the desired memory usage after aggregation. When the memory usage reaches m_{max}, the system will identify some clusters and merge all flows in one cluster to one metaflow, thus reduce the memory usage to m_{des}. It looks at every $L1$ cluster in the top list to find out if there is one or more $L2$ or $L3$ clusters inside this $L1$ cluster (e.g. $L2$ cluster B/E and $L3$ cluster D inside $L1$ cluster A in Fig. 1). A threshold r defines the minimum size (the number of flows in a cluster) of these identified $L2$ and $L3$ clusters. Among all identified clusters, the higher level ones have the higher priority to do aggregation. Among all clusters in the same level, the bigger ones have the higher priority. We choose these identified clusters one by one from high priority to low priority to do aggregation, until the memory usage is reduced to m_{des}. This simple flow aggregation algorithm can only meet some of the requirements discussed in Section III-B, such as keeping as many key attributes as possible. More details of this algorithm can be found in [28]. In the rest part of this section, we will introduce the entropy based flow aggregation algorithm, which meets all the objectives discussed in Section III-B.

B. Entropy

Entropy is a measurement of uncertainty of a random variable. Consider a random variable X that may take N_X discrete values. Suppose we randomly observe X for m times, then the empirical probability distribution on X is $p(x_i) = m_i/m, x_i \in X$, where m_i is the number of times we observe X taking the value x_i. The empirical entropy of X is then defined by Equation 1.

$$H(X) = -\sum_{x_i \in X} p(x_i)\ log\ p(x_i) \tag{1}$$

It is clear that $0 \le H(X) \le H_{max}(X) = log\ min\{N_X, m\}$. The above are standard results from [29]. We define the normalized entropy as $\widehat{H}(X) = \frac{H(X)}{H_{max}(X)}$. If $\widehat{H}(X) = 0$, then all observations of X are the same, i.e., $p(x_i) = 1$ for some $x_i \in X$. If $\widehat{H}(X) = 1$, then the observations have the highest degree of uncertainty or randomness, i.e., $p(x_i) = 1/min\{N_x, m\}$ for each observed x_i.

Now we introduce the concept of entropy to our problem of flow measurement. We first define several variables about the properties of flows in a given cluster:

- X: a random variable that denotes one of the four dimensions (srcIP, dstIP, srcPort and dstPort)
- $A = \{x_1, ..., x_n\}$: the set of distinct values in X (e.g., srcIP) that the observed flows take
- N: the total number of flows in the cluster
- N_i: the number of flows that take the value x_i
- $p_f(x_i) = N_i/N$: the empirical probability distribution of X (in terms of flows)
- $H_f(x_i)$: the empirical entropy of X (in terms of flows)
- $\widehat{H}_f(x_i)$: the normalized entropy of X (in terms of flows)
- B: the total number of bytes in the cluster
- B_i: the number of bytes that take the value x_i
- $p_B(x_i) = B_i/B$: the empirical probability distribution of X (in terms of bytes)
- $H_B(x_i)$: the empirical entropy of X (in terms of bytes)
- $\widehat{H}_B(x_i)$: the normalized entropy of X (in terms of bytes)

In the first part of our algorithm, we only look at the flow distribution in the given cluster. We use $p_f(x_i)$ as the probability distribution of X and $H_f(x_i)$ as the entropy of X. In the second part, we differentiate between big flows and small flows. We use $p_B(x_i)$ as the probability distribution of X and $H_B(x_i)$ as the entropy of X.

Consider the example in Fig. 1, 100 flows form a $L1$ cluster with a fixed srcIP. Then the entropy of dimension SrcIP is $\widehat{H}_f(X) = 0$ because all 100 observed flows have the same srcIP. Assume all flows have different srcPort, then its entropy with respect to number of flows is $\widehat{H}_f(X) = 1$. As to the dimension of dstPort, there are two significant values with the probability distribution of $p_f(1434) = 0.8$ and $p_f(80) = 0.1$. Assume the other 10 flows have different and unique dstPort, then the entropy of dimension dstPort is $\widehat{H}_f(X) \approx 0.2$. From this example one can conclude that entropies of these four dimensions are good indicators of their degree of uncertainty or randomness. It tells us if there are some significant values that stand out from others or all values are randomly distributed.

C. APP

We call those dimensions that have more than one value (e.g. the dstIP and srcPort of cluster B) *random dimensions*, and those dimensions which have one fixed value (e.g. the srcIP and dstPort of cluster B) *fixed dimensions*. When we merge all flows in a cluster into one metaflow, we only retain the fixed dimensions, and no longer keep track of the random dimensions. Among all clusters in Fig. 1, intuitively, we should choose cluster B to do aggregation because it meets most of the aggregation objectives. Firstly, cluster B contains enough flows compared with cluster D and E. Secondly, the degree of randomness of its random dimensions is large compared with cluster A. Thirdly, cluster B contains one more dimension (dstPort) than cluster A.

We use the Aggregation Priority Parameter (APP) to characterize these properties of cluster B. For the four dimensions of a cluster, assuming R = {all the random dimensions} and $F = R^c$ = {all the fixed dimensions}, then APP is defined by Equation 2.

$$APP_f = Min\{H_f(X_i)|X_i \in R\} \quad (2)$$

The larger the APP of a cluster, the higher priority this cluster would be given to be aggregated. The APP is a direct indicator of those clusters we want to aggregate because high APP means that the number of flows in the cluster is large, and that the entropy with respect to the individual random dimensions is large, and hence there are no individually significant flows with respect to these dimensions. Note that we use $H_f(X_i)$ rather than the normalized entropy $\widehat{H}_f(X_i)$ because we should choose the cluster that has enough flows.

What is the relationship between these two properties, degree of uncertainty and the number of flows in the cluster? Suppose there is a L_k cluster with a smaller L_{k+1} cluster inside it. The number of flows in the L_k (L_{k+1}) cluster is N_k (N_{k+1}), respectively. Assume everything is randomly distributed except there is a smaller L_{k+1} cluster inside the big L_k cluster. Then the APP values of the L_{k+1} and the L_k clusters are,

$$\begin{aligned} &APP_f(k+1) = log\ N_{k+1} \\ &APP_f(k) = -\tfrac{N_{k+1}}{N_k} log \tfrac{N_{k+1}}{N_k} - \tfrac{N_k - N_{k+1}}{N_k} log \tfrac{1}{N_k} \end{aligned} \quad (3)$$

We would choose the L_{k+1} cluster if $APP_f(k+1) > APP_f(k)$. Given a N_k, we can get a threshold θ from Equation 3. If $N_{k+1}/N_k > \theta$, then $APP_f(k+1) > APP_f(k)$. For example, when $N_k = 1000$, $\theta = 0.25$. Hence the smaller cluster would be chosen if the size ratio $N_{k+1}/N_k > 0.25$. In other words, when the size of the smaller cluster reaches 0.25 of the bigger cluster, it is too significant to be ignored. The threshold θ decreases as the value of N_k increases. In Fig. 1, the size ratio of cluster B and cluster A is 0.8, so it is too significant to be ignored.

As we have mentioned in Section III-B, if there are several big flows in the identified cluster, we would pick them out. To do this, we further define APP (in terms of bytes) as:

$$APP_B = Min\{H_B(X_i)|X_i \in R\} \quad (4)$$

High APP_B means there will be no flow much larger than other flows in the identified cluster. Now we assume all flows in cluster B have the same size except flow C, whose size is 10 times of that of other flows. Cluster D has 10 flows with the same dstIP. Then $APP_B(B) = 5.73$. If we pick out flow C and cluster D, the APP of cluster $F := B - C - D$ is $APP_B(F) = 6.11$. Among all clusters in Fig. 1, cluster F has the highest APP_B. Under this condition, we would choose cluster F to do aggregation. So what the flow aggregation algorithm should do is to find out cluster F among all sub-clusters of cluster A.

D. Algorithm description

As mentioned in Section IV, we maintain a top list for the IP addresses with the largest flow numbers. From the top list, now we have several big $L1$ clusters with fixed srcIP or dstIP. The entropy based flow aggregation algorithm is to find out every $L1$ cluster's sub-clusters that have the largest APP, just as finding out sub-cluster F in $L1$ cluster A. We call them *desired sub-clusters*. These desired sub-clusters could not be subordinate to or overlap with each other. Among them, the sub-cluster whose APP is the largest will be chosen. However, if several sub-clusters do not contain or overlap with each other (we call them *distinct clusters*) and have similar APP, they would all be identified.

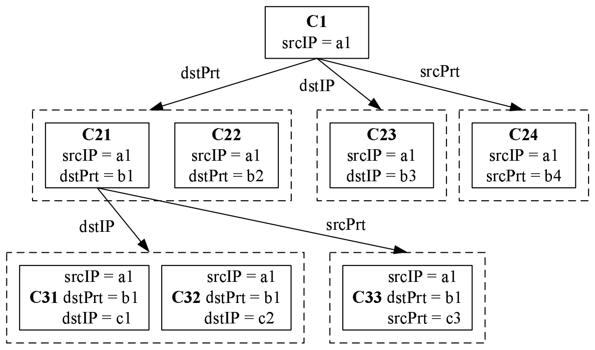


Fig. 3. An example of the procedure for finding out desired sub-clusters.

1) **First Step: find out the sub-clusters with the highest APP_f:** The first part of this algorithm iteratively computes APP_f for each level of clusters and finds out those sub-clusters with the highest APP_f, just as finding out cluster B in Fig. 1. Fig. 3 illustrates an example of this procedure. The procedure starts from a $L1$ cluster $C1$ with a fixed srcIP, whose three random dimensions are sorted by their entropy in ascending order, dstPort, dstIP and srcPort. The reason for this reordering is that the lower the entropy of one dimension, the higher the possibility that there is a big sub-cluster in this dimension. For example, in Fig. 1, dimension dstPort has a big sub-cluster B. Looking at dimension dstPort given srcIP = a1, we find out all $L2$ clusters such as $C21$ and $C22$ whose flow numbers are greater than a threshold f_r. If any one of these $L2$ clusters has a larger APP_f than its parent cluster $C1$, this $L2$ cluster will replace cluster $C1$ to become one of the candidates of the desired clusters. We also zoom in on the $L3$ sub-clusters of each $L2$ cluster. Take $C21$ as an example, it has two random dimensions sorted by entropy. We find out those $L3$ clusters such as $C31$, $C32$ and $C33$, whose flow numbers are greater than f_r, and compute APP_f of these clusters. If any one of these $L3$ clusters has a larger APP_f than its parent and the $L1$ cluster, it will replace them to become one of the candidates. After examining all the sub-clusters in the three random dimensions, we get a list of candidates for the desired sub-clusters, which are distinct clusters with similar APP_f.

The complexity of this procedure largely depends on the threshold f_r. The higher this threshold, the less sub-clusters we need to examine. To dynamically adjust f_r, we maintain an APP_{max} which represents the current maximum APP_f value of all clusters that we have examined. We have noted that the maximum APP_f of a cluster equals to $log\ N$, where N is the number of flows in this cluster. We can set $f_r = 2^{APP_{max}}$ and only examine those sub-clusters whose number of flows greater than f_r, because only these sub-clusters have the possibility to have APP_f values larger than APP_{max}. To finally find out several distinct clusters with similar APP_f, we can set $f_r = k * 2^{APP_{max}}$, where $0 < k < 1$.

Algorithm 1 finding out sub-clusters

```
Input:  Cluster $C(D)$, where $D = \{srcIP\}$ or $\{dstIP\}$
Output:  sub-clusters of high $APP$: $CList$
FindSubCluster $(C)$\{
1.   compute $C.APP_f$
2.   initialize $CList$, $APP_{max}$, $f_r$
3.   $R = T \backslash D$, where $T = \{srcIP, dstIP, srcPort, dstPort\}$
4.   sort $R$ to $R = \{r_i | H(r_0) < H(r_1) < H(r_2), 0 \le i \le 2\}$
5.   for $i$ = 0 to 2
6.      $L2 = \{S_k | S_k(D \cup \{r_i\}), S_k.N > f_r\}$
7.      for every $S_k$ in $L2$
8.         compute $S_k.APP_f$
9.         Update $APP_{max}$, $f_r$
10.        $G = R \backslash \{r_i\}$
11.        sort $G$ to $G = \{g_j | H(g_0) < H(g_1), 0 \le j \le 1\}$
12.        for $j$ = 0 to 1
13.           $F = D \cup \{r_i\} \cup \{g_j\}$
14.           $L3[j] = \{SS_l | SS_l(F), SS_l.N > f_r\}$
15.           for every $SS_l$ in $L3[j]$
16.              compute $SS_l.APP_f$
17.              Update $APP_{max}$, $f_r$
18.           end for
19.        end for
20.        $list[i]$.append ( MaxAPP $(S_k, L3[0], L3[1])$ )
21.     end for
22.     $CList$ = MaxAPPDistinctCluster $(CList, list[i])$
23.  end for
24.  for every $C_P$ in CList
25.     GetMaxEntropySubset $(C_P)$
26.  end for
27.  reorganize $(CList)$
}
```

Algorithm 1 represents the algorithm for finding out desired sub-clusters from the original $L1$ cluster. The input to the function is an $L1$ cluster C with fixed dimension D. The output of the function is $CList$, a list of sub-clusters with the largest APP. Line 1 to 23 of the function describe the first step of the algorithm. Line 1 to 4 are the initialization steps including setting the initial value of APP_{max} to $C.APP_f$, computing corresponding f_r, setting $CList$ to C, and sorting R by entropy in ascending order, where R is the three random dimensions of C. As shown in line 5 to 23, we examine the three random dimensions one by one to get a list of candidates with the highest APP_f. First, in line 6, all $L2$ sub-clusters S_k are put into $L2$. These S_k have fixed dimensions $D \cup \{r_i\}$ and have flow numbers greater than f_r. The two random dimensions of S_k are represented by G. In line 7 to 21, every S_k is zoomed in on its $L3$ sub-clusters $L3[j]$, which have three fixed dimensions represented by F. In line 20, the $L2$ sub-cluster S_k, or one or several $L3$ sub-clusters from $L3[0]$, or from $L3[1]$ are chosen and appended to $list[i]$, depending on which one has the highest APP_f. In line 22, sub-clusters in $list[i]$ are appended to $CList$ as long as they are distinct clusters and have similar APP_f with the highest one.

2) **Second Step: pick out big flows from the candidates:** Till now we do not differentiate between big and small flows. The second step is to pick out big flows from these candidates, such that the remaining subset has the highest APP_B, e.g., cluster F in Fig. 1. We call it the *maximum entropy subset*. After we extract the maximum entropy subset from each candidate, they will have the new APP_B, which are different from the original APP_f. At last, we reorganize the list by removing those sub-clusters whose new APP_B are not at the same level with the sub-cluster having the highest APP_B. This part is shown in line 24 to 27 of Algorithm 1.

Algorithm 2 finding out the maximum entropy subset

```
Input:   cluster C with random dimensions rd
Output: maximum entropy subsets of cluster C
GetMaxEntropySubset(C, rd) {
1.  d = MinEntropy(C,rd)
2.  sort C to $\{S_i | S_i(d), B_{i-1} < B_i, 1 \le i \le X_d\}$
3.  $E = 0$; $H_k = 0$; $k = 0$; $R = 0$
4.  for $i = 1$ to $X_d$
5.     $P_i = B_i / \sum_{j=1}^{X_d} B_j$
6.     $R = R + P_i$
7.     $E = E - P_i log P_i$
8.     $H' = E/R + logR$
9.     if $(H' > H_k)$ $H_k = H'$; $k = i$
10. endfor
11. return $k$, $H_k$
}
```

We use Algorithm 2 to find the maximum entropy subset of a cluster C. First we compute H_B for every random dimension of C, and set d to the dimension with the minimum entropy (line 1). After that, all flows in C are sorted to $\{S_i | S_i(d), B_{i-1} < B_i, 1 \le i \le X_d\}$ (line 2). X_d is the number of different values in dimension d. S_i is the set of flows which have the same value in dimension d. Let B_i be the total number of bytes of all flows in S_i, and these S_i are sorted in ascending order of B_i.

The entropy of dimension d of cluster C is $H_B = -\sum_{i=1}^{X_d} P_i log P_i$, where $P_i = B_i / \sum_{i=1}^{X_d} B_i$. We want to pick out some biggest $\{S_i, k+1 \le i \le X_d\}$ and let the remaining $\{S_i, 1 \le i \le k\}$ be the maximum entropy subset. Sometimes the maximum entropy subset is generated by picking out the smallest S_i. We do not consider this condition because we only need to pick out big flows. Let $R = \sum_{i=1}^{k} P_i$, then the entropy of $\{S_i, 1 \le i \le k\}$ is $H_k = -\sum_{i=1}^{k} (P_i/R) log(P_i/R) =$

$E_k/R + logR$, where $E_k = -\sum_{i=1}^{k} P_i log P_i$ is the part that the subset k contributes to H_B. Having this relationship, we can find out the maximum entropy subset by going through these S_i only once, as described in line 3 to 10. The output are k and H_k, that is to say, the maximum entropy subset is $\{S_i, 1 \leq i \leq k\}$ and its entropy is H_k.

E. Flow aggregation and export

After the algorithm identifies the desired sub-clusters, the system merges all flows in one desired sub-cluster to one metaflow. If there are several desired sub-clusters with similar APP, we get one metaflow from each sub-cluster. If the desired sub-cluster is like cluster $F := B - C - D$ in Fig. 1, then all flows except flow C that are in cluster B but not in cluster D are merged to one metaflow, while flow C and the flows in cluster D are not modified.

The metaflow keeps the values of fixed dimensions of the cluster and set the values of random dimensions to *, denoting all possible values. Other attributes of this metaflow are similar to those defined in [8]: the packet/byte count is the sum of the number of packets/bytes of all aggregated flows, the timestamp of the first packet (create time of the metaflow) is the minimum of this timestamp of all aggregated flows, and the timestamp of the last packet (modify time of the metaflow) is the maximum of this timestamp of all aggregated flows.

When a packet arrives, the system determines if this packet belongs to an active flow. For a metaflow, only fields of an exact value are compared with corresponding fields of the packet. For example, if a metaflow is (srcIP = *, dstIP = 210.0.0.3, srcPort = *, dstPort = 80,TCP), then all following packets of web traffic to the server with IP address of 210.0.0.3 will be regarded as belonging to this metaflow. The metaflow will be terminated and exported as other normal flows when the termination criteria are met, including *inactive timer* and *active timer*. Note that the criteria based on certain TCP flags would not be used, because these flags indicate the termination of only one flow but not the metaflow.

When new packets do not belong to any active flow but belong to one metaflow, the number of packets and bytes of this metaflow will be updated. So we can get accurate packet and byte counts for the metaflow. The number of flows of the metaflow cannot be counted directly because we must distinguish between packets belonging to the old and the new flows and increment the flow counter only if the flow is new. We use the multi-resolution bitmap algorithm which was proposed in [30] to estimate the number of flows. Before merging flows in one cluster to a metaflow, the system creates a multi-resolution bitmap, and maps all flow IDs in this cluster to the bitmap. Whenever a new packet is determined to belong to this metaflow, the system will map its corresponding flow ID to the bitmap. We can get quite accurate result for the estimated number of flows if we use a large enough bitmap. Because there will not be too many identified clusters, the memory requirement and the processing overhead are acceptable.

VI. Analysis

In this section, we analyze the algorithm proposed in this paper (*entropy based flow aggregation*), and compare it with other solutions including, 1) NetFlow without memory constraint (*basic NetFlow*), 2) NetFlow which rejects new flows when the cache is full (*rejecting NetFlow*), 3) NetFlow which exports more aggressively when the cache is full (*exporting NetFlow*), 4) *adaptive NetFlow* (proposed in [5]) that adapts the sampling rate to traffic, and 5) the *simple flow aggregation* algorithm proposed in [28]. We take the implementation of fprobe as an example of *basic NetFlow*.

A. Resource requirement

First we analyze the resources required by the algorithms. The key resource measures include the size of flow memory, the size of export bandwidth, and CPU utilization.

1) **Flow memory:** Because of our modified data structure, our algorithm uses a bit more memory than *basic NetFlow*. Assume S_f is the size of a flow entry, S_{ip} is the size of an IP Node in Fig. 2. Considering the worst case, every flow entry has different srcIP and dstIP, then our algorithm uses $(S_f + 2 * S_{ip} + 4)/S_f$ times memory of *basic NetFlow*. 4 denotes we use one more pointer in the flow entry. S_f is around 64 bytes, S_{ip} is around 10 bytes (two pointers and one counter). So our data structure uses 1.4 times the memory of *basic NetFlow* in the worst case.

Besides the memory used for storing active flows, *entropy based flow aggregation* uses additional memory when it does flow aggregation. The first one is the temporary memory used in identifying the desired sub-clusters for each $L1$ cluster. Assume the $L1$ cluster has N flows, the system uses a linked list to store the information of all the flows that belong to this cluster, which includes a pointer to this flow. Because we need to sort all the flows in this cluster when we compute entropy, we use this linked list to store the sorted flow information. Assume the size of a node in this linked list is S_{fi}, then the total memory needed for each cluster is $N * S_{fi}$. This memory will be freed after we identify the desired sub-clusters in this cluster, so the peak memory is $N_{max} * S_{fi}$, where N_{max} is the maximum flow number of all the $L1$ clusters. S_{fi} can be 8 bytes if each node in the linked list only stores a pointer to the flow and a pointer to the next node. The second one is the memory used in the bitmap algorithm for counting the flow number in every metaflow, which is $O(log(N_{meta}))$, where N_{meta} is the flow number in each metaflow. This part of memory is not freed until this metaflow is terminated and exported.

Adaptive NetFlow may also use more memory than *basic NetFlow*. The algorithm divides the NetFlow operation into measurement bins. A fixed size of the measurement bin could be a problem, because its optimal size depends on the traffic mix. If the measurement bin is too large, it keeps many short flows unnecessarily long in the memory cache, and uses more memory than necessary. If the memory is bounded, then the adaptive algorithm decreases the sampling rate lower than necessary, and sacrifices the accuracy of all flows. On the other hand, if the measurement bin is too small, it splits many long flows to several flows, hence increases the export bandwidth and burdens the collector. Once *adaptive NetFlow* fixes the size of the measurement bin, how much memory that it uses more than *basic NetFlow* depends on the traffic mix.

2) **Export bandwidth:** Besides memory, another main resource constraint is export bandwidth. Our algorithm uses either the same or less export bandwidth than *basic NetFlow*. Its export bandwidth is the same as *basic NetFlow* when the system does not aggregate flows, and less than *basic NetFlow* when it performs aggregation. *Exporting NetFlow* may use a very high export bandwidth, and may flood the collector. In *adaptive NetFlow*, a router operator specifies the reported number of flow records M desired for each measurement bin, the algorithm guarantees this fixed export bandwidth by decreasing the sampling rate.

3) **CPU utilization:** We first describe the overhead to normal flow operations, that is, update the flow cache when new packets come in and periodically check the flow cache looking for expired flows. In extreme conditions, if a large part of flows have the same srcIP or dstIP, then the corresponding IP node list will be so long that it would slow down flow lookup. Actually we can define a threshold, and when the length of the IP node list reaches this threshold it triggers aggregation. As mentioned in Section IV, we maintain a top list for the IP addresses with the largest flow numbers. Another overhead to normal flow operations is to maintain this top list. Every time we create or delete a flow entry, we need to update the top list. However, the maximum number of the top list is not large (20 or even less is enough). In addition, We need some extra processing to find out the desired sub-clusters for every $L1$ cluster. The complexity of this algorithm is $O(N^2)$, where N is the number of flows in the $L1$ cluster. The detailed complexity analysis is given in Appendix A.

B. Accuracy

Various network anomalies all tend to generate excessive number of flows, often exceeding the resource contraints of traffic monitors. Most countermeasures need to give up some accuracy in traffic capturing. For example, one countermeasure is *rejecting NetFlow* that rejects all new flows when the cache is full. Another countermeasure is *adaptive NetFlow* that automatically chooses a lower sampling rate during a DoS attack. While this measure degrades the system gracefully during attack, it unfortunately affects the accuracy of all flows collected. Sometimes the bottleneck is not the netflow (in the router), but the flow export process, especially in *exporting NetFlow*. Accuracy can be lost in two ways: (a) routers export NetFlow records to the collector using UDP and flow records are lost due to congestion; (b) the post-processing analysis and visualization tools cannot keep up with this avalanche of flows.

When comparing with those countermeasures that lose flows during heavy load, the superiority of flow aggregation is easily established. The comparison of flow aggregation against *adaptive Netflow*, however, is hard to quantify. By lowering the sampling rate, *adaptive Netflow* will lower the accuracy of all flows with equal probability; hence all kinds of aggregates (by ports, IP addresses, etc.) also lose accuracy proportionally. On the other hand, flow aggregation uses a lower resolution only for some, but not all clusters, so loss of accuracy for different aggregates is quite different. The loss of accuracy brought by flow aggregation depends on how aggregation is performed and whether the network operators care about the details lost during aggregation. If the dimensions that we discard during aggregation are included in the dimensions network operators are interested, then there is loss of accuracy; otherwise, there is effectively no loss of accuracy. For example, if we identify and aggregate a cluster of fixed dstIP plus dstPort, then we still get accurate results for protocol and application breakdowns, and the destination host. However, if the network operators are interested in the srcIP or srcPort of this traffic, they cannot get the accurate statistics of these two dimensions. We compare the accuracy of *entropy based flow aggregation* with other solutions by experiments, described in Section VII.

C. Practical Considerations

Often the reason for abnormal traffic conditions is due to security attacks and such attacks often have some common patterns. So our algorithm can relieve the resource overload by identifying these traffic clusters and aggregating these large amounts of short flows into a few flows. Sometimes, the overload may be caused by undifferentiated traffic not dominated by any particular cluster, e.g., a shift in load caused by link failure or routing change. In this situation, even if we aggregated all $L1$ clusters, the memory which will be freed may still not satisfy the requirement. In other words, our solution cannot deal with this case. From this point of view, our solution should be considered as a way to complement other current solutions, rather than completely replace them. If our algorithm fails to find appropriate clusters, we conclude that the traffic is undifferentiated and take other actions such as in *rejecting NetFlow*, *exporting NetFlow* or *adaptive NetFlow*.

Another problem is that the traffic from a busy web server may be identified as the desired cluster by our algorithm. So if there are links in the network that are dominated by particular clusters in the normal case, network operators can use policy to protect such clusters, resulting in the algorithm looking for other clusters or performing aggregation only when they exceed their policy defined limits. If there is a flash crowd to a server, it will be very similar to a DoS attack from the point of view of our system. Then the flows of this flash crowd will be identified and aggregated when they exceed the limits defined by network operators.

VII. Experimental evaluation

In this section, we evaluate different solutions by running them on synthetic and real trace files. These solutions include *basic NetFlow*, *rejecting NetFlow*, *exporting NetFlow*, *adaptive NetFlow*, *simple flow aggregation* and *entropy based flow aggregation*. We first present our experimental setup, and then give out evaluation results on different trace files.

A. Experimental setup

We first present our metrics and experimental datasets. For a given cluster, assume n_f, n_p, n_B are the number of flows, packets and bytes of this cluster. This given cluster can be any traffic aggregate that network administrators are interested in,

	t	τ	$[l,\ h]$	$[T_s,\ T_e]$	Description
A	10s	1s	[900, 1200]	[0, 5400s]	
B	10s	5s	[180, 240]	[0, 5400s]	
C	10s	1s	[180, 240]	[0, 5400s]	
D	10s	5s	[36, 48]	[0, 5400s]	
E	0.1s	0.1s	[2, 20]	[2700s, 3700s]	DoS attack
F	0.1s	0.1s	[2, 20]	[2000s, 4000s]	worm spreading
G	10s	1s	[180, 240]	[0, 5400s]	web traffic

TABLE II
Synthetic trace flow information.

e.g., all the traffic sent from a specific host. *Basic NetFlow* can get accurate values of these numbers, while the estimated values from other solutions will be different from the accurate ones. Different solutions use different amounts of resources and have different accuracy. We use the following metrics to evaluate these solutions:

- memory usage - memory used at the observation point
- export bandwidth - number of flows exported during the past 2 minutes
- relative error - average error for byte, packet, or flow estimates

$$relerr_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\frac{\widehat{n_j^i}}{n_j}-1)^2}, 1 \le i \le m, j = f, p, B \quad (5)$$

Equation 5 is used to compute the relative error of a given solution. We repeat the experiment for m times, $\widehat{n_f^i}$, $\widehat{n_p^i}$, and $\widehat{n_b^i}$ are the estimated value for the number of flows, packets and bytes in the i^{th} experiment of this solution.

The data sets that we measure different solutions are:

- "Synthetic" - a synthetic trace file generated by CSIM
- "DarpaIDE" - the training data of the 1998 DARPA Intrusion Detection Evaluation
- "CaidaOC48" - a 30 minute trace of the traffic on an OC48 IP backbone link, provided by Caida

B. Resource evaluation on synthetic trace file

We use CSIM, a general purpose discrete-event simulator, to generate a synthetic trace file. During the observation time of 5400s, there are seven types (A, B, C, D, E, F, G) of flows. Flows of each type arrive as a Poisson process, and the inter flow time is exponentially distributed with mean t_i. In every flow, the packet arrival is also Poisson, and inter packet time is exponentially distributed with mean τ_i. The number of packets for every type of flow is uniformly distributed in a range $[l_i, h_i]$. The characteristics of these seven types of flows are shown in Table II. Flow E is a simulated DoS attack, all flows of type E have the same dstIP and dstPort. It does not last during the whole duration of 5400s, but starts at 2700s and ends at 3700s. Flow F is a simulated worm spread, all flows of type F have the same srcIP. It starts at 2000s, and ends at 4000s. Flow A, B, C, D and G are simulated normal traffic, they last during the whole duration. t, τ, l and h are different for each type, so they have different characteristic, long-lived or short-lived, dense or sparse. But compared with flows E and F, their t and τ are longer, l and h are larger. Their IP address and port are randomly generated except that all flows of type G are web traffic to the same dstIP.

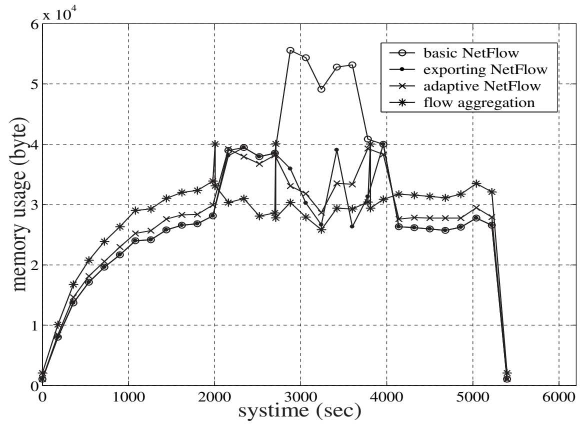


Fig. 4. Memory usage for different solutions on synthetic trace.

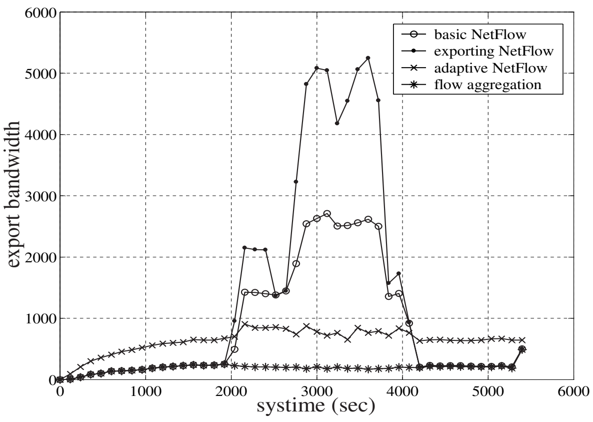


Fig. 5. Export bandwidth for different solutions on synthetic trace.

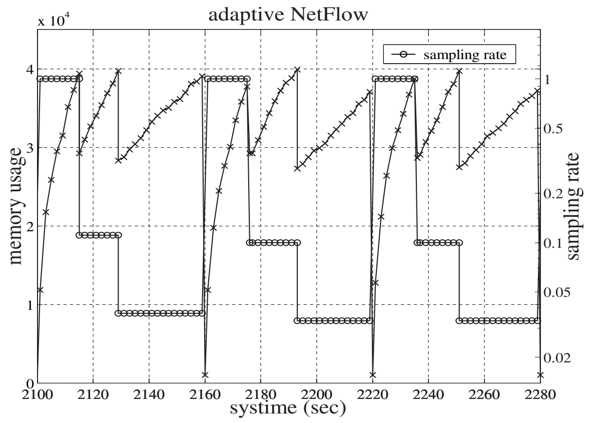


Fig. 6. Memory usage and sampling rate in several measurement bins of *adaptive Netflow* on synthetic trace.

Fig. 4 shows the memory usage of different solutions. we define $m_{max} = 40000\ bytes$, and $m_{des} = 30000\ bytes$. When memory usage reaches m_{max}, different systems (except *basic NetFlow*) perform different operations to reduce memory usage to m_{des}, while *basic NetFlow* is assumed to have unlimited memory. Fig. 5 shows the export bandwidth of these solutions. We record export bandwidth every 2 minutes, which is defined as the number of flows exported during the past 2 minutes. For *exporting NetFlow*, before reaching m_{max}, its memory usage and export bandwidth are the same as that of *basic NetFlow*. After exceeding m_{max}, its memory usage is bounded by m_{max}, but the export bandwidth is much higher than that of *basic NetFlow*.

For *adaptive NetFlow*, we use the measurement bin of 1 minute. Before reaching m_{max}, memory usage of *adaptive NetFlow* is a little greater than that of *basic NetFlow*, due to the unnecessarily long time that *adaptive NetFlow* keeps short flows in the memory, as we mentioned in Section VI-A1. On the other hand, export bandwidth of *adaptive NetFlow* is also greater than that of *basic NetFlow*. The reason is that many flows we generated are much longer than the measurement bin of 1 minute, so they are split into several flows. After exceeding m_{max}, its memory usage is bounded by m_{max} and the export bandwidth is stable. For more detail, its memory usage and sampling rate in several measurement bins are shown in Fig. 6. At the beginning of one measurement bin, the sampling rate is equal to 1 (process every packet). When the memory usage reaches m_{max}, *adaptive NetFlow* decreases its sampling rate. At the end of one measurement bin, all active flows in the cache memory are exported and the sampling rate is reset to 1. In this experiment, the sampling rate decreases to a low value of around 1/30 (as shown in Fig. 6).

For *entropy based flow aggregation*, before reaching m_{max}, its memory usage is larger than that of *basic NetFlow*, due to the overhead caused by the new data structure, as we analyzed in Section VI-A1. Its export bandwidth is the same as that of *basic NetFlow*. At around 2000 sec, the memory usage exceeds m_{max}. The algorithm identifies the cluster of the simulated worm spread (with the same srcIP) and aggregates flows in this cluster. Both the memory usage and export bandwidth are much lower than those of *basic NetFlow*. At around 2700 sec, the simulated DoS attack is generated, so the memory usage exceeds m_{max} again, which triggers the second aggregation. The third aggregation occurs at around 3800 sec. The reason is that we use an *active timer* of 30 minutes, so the metaflow generated from aggregation at 2000 sec is terminated and exported at 3800 sec. But because packets in this worm spread have not stopped, many new generated flows make the memory usage reach m_{max} again and trigger the third aggregation. At the time when system performs aggregation (around 2000 sec, 2700 sec, 3800 sec), the peak memory usage is a little higher than m_{max}, which includes the additional temporary memory used in identifying the desired sub-clusters.

C. Accuracy evaluation on "DarpaIDE" dataset

In this section, we will show results from experiments on traces of actual traffic. The dataset we use is part of the training data of the 1998 DARPA Intrusion Detection Evaluation [31], which contained a wide variety of simulated intrusions. We choose Wednesday data of week 1 as our experiment data, because it contains DoS attacks such as Smurf. For brevity, we omit the resource evaluation results, which are the same as what we expect and similar to those of the "Synthetic" dataset. To compare the accuracy of *adaptive NetFlow* and *entropy based flow aggregation*, we perform post-processing on the flow records exported from *adaptive NetFlow*, *entropy based flow aggregation* and *basic NetFlow*. We perform three post-processing steps based on the applications used by most analysis and visualization tools.

The first post-processing step is protocol breakdown. For these solutions, protocol breakdown counts the number of bytes, packets and flows for TCP, UDP and ICMP. We repeat each experiment for 5 times, and get *relerr* using Equation 5. *Relerr* results for *adaptive NetFlow* and *entropy based flow aggregation* are shown in Table III. It may be unfair to compare the *relerr* results for the number of flows directly, because we use the bitmap algorithm to count the number of flows of the identified clusters. We also give out the flow error result without using the bitmap algorithm, as shown in the "flow err" column. The flow error result using the bitmap algorithm is shown in the "bitmap" column.

The second post-processing step is port breakdown, which counts the number of bytes, packets and flows for different ports. For *adaptive NetFlow* and *entropy based flow aggregation*, we calculate *relerr* for the top srcPort/dstPort sorted by the number of bytes, packets and flows. For brevity, we only show *relerr* of the top 8 srcPorts sorted by the number of bytes in Table IV, and omit the other five *relerr* tables. The third post-processing step is to find the top hosts by bytes, packets or flows of traffic generated/received. *Relerr* results of top 8 dstIP sorted by bytes are shown in Table V.

From these *relerr* results, we conclude that *entropy based flow aggregation* provides better accuracy for legitimate flows than *adaptive NetFlow*. As shown in these three tables, *entropy based flow aggregation* achieves accurate results with zero byte errors and packet errors. The reason is that flow aggregation keeps the accurate byte and packet counts for metaflows. The Smurf attack in this dataset generated large number of ICMP flows, which causes the memory usage to reach m_{max} and triggers flow aggregation. The TCP and UDP traffic is not affected, so there is no error in the flow counts for TCP and UDP in Table III and the flow counts for all the top srcPorts in Table IV. On the other hand, we aggregate those ICMP flows to the victim dstIP in the Smurf attack into one metaflow and do not keep the flow counts, so the flow counts for ICMP and the victim dstIP have high error rates of 66.04% and 57.60% respectively. After using the bitmap algorithm, the flow count results become much more accurate, as shown in the "bitmap" columns.

D. Accuracy evaluation on "CaidaOC48" dataset

The "CaidaOC48" data set is a 30 minute trace from Aug 2002 of one direction of traffic on an OC48 link located in San Jose, provided by Caida. The flow rate, packet rate, and byte rate of this data set is 5k/s, 75k/s, and 396M/s respectively. We artificially generate several DoS attacks and worm spreads and mix it with "CaidaOC48". The information of these attacks is shown in Table VI. The time of traffic A being [120s, 180s]

adaptive NetFlow					
protocol	%	byte err	packet err	flow err	bitmap
TCP	85.2	0.21	0.28	15.18	NA
UDP	0.6	0.97	0.77	33.15	NA
ICMP	14.2	21.21	21.04	36.99	NA
entropy based flow aggregation					
protocol	%	byte err	packet err	flow err	bitmap
TCP	85.2	0	0	0	0
UDP	0.6	0	0	0	0
ICMP	14.2	0	0	66.04	0.99

TABLE III
Relative error (%) of protocol breakdown on "DarpaIDE" dataset.

adaptive NetFlow					
srcPort	%	byte err	pkt err	flow err	bitmap
80 , tcp	66.54	0.31	0.32	16.63	NA
20 , tcp	11.45	0.26	0.26	8.31	NA
25 , tcp	0.58	0.68	0.36	3.00	NA
53 , udp	0.52	1.73	1.26	26.68	NA
21 , tcp	0.075	1.29	0.40	21.31	NA
23 , tcp	0.072	2.05	1.27	16.19	NA
123 , udp	0.069	2.90	2.90	37.91	NA
11306 , tcp	0.019	0	0	0	NA
entropy based flow aggregation					
srcPort	%	byte err	pkt err	flow err	bitmap
80 , tcp	66.54	0	0	0	0
20 , tcp	11.45	0	0	0	0
25 , tcp	0.58	0	0	0	0
53 , udp	0.52	0	0	0	0
21 , tcp	0.075	0	0	0	0
23 , tcp	0.072	0	0	0	0
123 , udp	0.069	0	0	0	0
11306 , tcp	0.019	0	0	0	0

TABLE IV
Relative error (%) of port breakdown on "DarpaIDE" dataset.

adaptive NetFlow					
dstIP	%	byte err	pkt err	flow err	bitmap
172.16.114.50	14.66	21.17	19.68	29.74	NA
172.16.116.44	9.09	0.46	1.21	14.11	NA
172.16.114.169	8.16	0.44	0.99	12.24	NA
172.16.114.148	5.19	0.95	0.62	17.00	NA
172.16.113.84	5.03	0.97	1.23	9.08	NA
172.16.114.207	4.64	1.32	1.32	16.94	NA
172.16.112.194	4.44	1.31	1.32	15.62	NA
172.16.112.149	3.88	0.97	1.05	6.33	NA
entropy based flow aggregation					
dstIP	%	byte err	pkt err	flow err	bitmap
172.16.114.50	14.66	0	0	57.60	0.86
172.16.116.44	9.09	0	0	0	0
172.16.114.169	8.16	0	0	0	0
172.16.114.148	5.19	0	0	0	0
172.16.113.84	5.03	0	0	0	0
172.16.114.207	4.64	0	0	0	0
172.16.112.194	4.44	0	0	0	0
172.16.112.149	3.88	0	0	0	0

TABLE V
Relative error (%) of IP breakdown on "DarpaIDE" dataset.

means it starts at 120s and ends at 180s. SrcIP of traffic A being "*.*.*.*" means its srcIP is a randomly chosen IP address. Byte of traffic A being 40 means its packet size is 40 bytes per packet. SrcIP of traffic B being "3 hosts: a.b.c.d" means there are three hosts that send the traffic, and dstIP of traffic B being "a.b.*.*" means the first two parts of the dstIP are equal to the srcIP.

As we have mentioned in Section I, for high-speed interfaces, Cisco introduced sampled NetFlow. We do not use sampling for the last two experiments because of their low data rate (about 118 Kbytes/sec for the synthetic dataset and 4 Kbytes/sec for the "DarpaIDE" dataset). For this OC48 data set, we set the packet sampling rate to 1/100. That is, we use a basic sampling rate for all solutions including *basic NetFlow*. In addtion, we only focus on the error caused by decreasing the sampling rate or performing flow aggregation under memory shortage and ignore the error caused by the packet sampling under normal conditions. So we preprocess the trace file by sampling it using a sampling rate of 1/100. After that we run different solutions on the pre-sampled trace file.

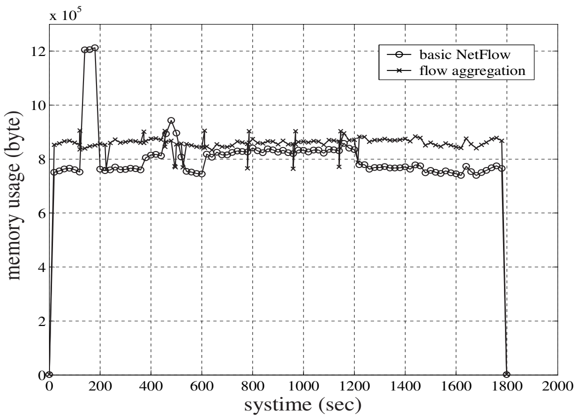


Fig. 7. Memory usage for *entropy based flow aggregation* on "CaidaOC48" dataset.

We set $m_{max} = 900000\ bytes$ and $m_{des} = 850000\ bytes$ for all solutions. For *adaptive NetFlow*, we use 20 sec as the size of the measurement bin such that its export bandwidth is similar with that of *entropy based flow aggregation*. For *simple flow aggregation*, we set $r = 30$, which means the minimum size for the identified clusters is 30. Fig. 7 is the memory usage of *entropy based flow aggregation*. The increases in the memory usage of *basic NetFlow* are caused by all the DoS attacks and worm spreads shown in Table VI except traffic F, whose flow rate is too low to trigger flow aggregation.

Entropy based flow aggregation accurately identifies all the clusters of the DoS attacks and worm spreads except traffic F. The metaflows resulted from the *entropy based flow aggregation* present the network administrators some useful information about these DoS attacks and worm spreads. For example, for traffic B and C, *entropy based flow aggregation* generates three metaflows, each one corresponds to one host that sends out the worm traffic. Each metaflow gives out the information including begin time, duration, srcIP, dstPort, protocol, byte number, packet number and flow number of this

	time	simulated attack	flow rate	srcIP	dstIP	srcPort	dstPort	protocol	byte
A	[120s, 180s]	DDoS	33k/s	*.*.*.*	162.131.189.129	*	80	TCP	40
B	[360s, 480s]	Blaster worm	0.75k/s	3 hosts: a.b.c.d	a.b.*.*	1000 - 1999	135	TCP	40
C	[360s, 480s]	Blaster worm	2.25k/s	3 hosts: a.b.c.d	*.*.*.*	1000 - 1999	135	TCP	40
D	[450s, 510s]	DoS after Blaster	10k/s	a.b.*.*	207.46.18.94	1000 - 1999	80	TCP	40
E	[600s, 1200s]	Slammer worm	5k/s	5 hosts: x.y.z.w	*.*.*.*	3355	1434	UDP	376
F	[1620s, 1800s]	Welchia worm	0.5k/s	239.187.123.15	239.187.*.*			ICMP	92

TABLE VI
Information of simulated DoS attacks and worm spreads.

worm spreading. DstIP and srcPort of the worm traffic are not given by the metaflow because of flow aggregation, which are randomly chosen by the worm.

In accuracy evaluation on the "DarpaIDE" dataset as shown in Section VII-C, we only compare the accuracy of *adaptive NetFlow* and *entropy based flow aggregation*. Because the flow rate is very low, there is very few other legitimate traffic when the DoS attacks occur. So *simple flow aggregation* acts in a similar way as *entropy based flow aggregation* does, and *rejecting NetFlow* also has a good result because most of the packets that are thrown away are DoS attack packets. On the other hand, the "CaidaOC48" dataset has a high byte/flow rate, so we compare the accuracy of all the four solutions for the "CaidaOC48" dataset.

$Relerr$ results of some top dstIPs of these solutions are shown in Table VII. The byte error and flow error of *rejecting NetFlow* are similar, because it just rejects new flows when the flow cache is full and does nothing to the lost flow data. On the other hand, the flow error of *adaptive NetFlow* is much greater than its byte error, because *adaptive NetFlow* decreases the sampling rate and compensates this by multiplying the result by the sampling rate while cannot do corresponding compensation to the flow numbers. Our two flow aggregation solutions provide better accuracy than the other two solutions. For *entropy based flow aggregation*, only the victim of traffic A has error in the flow count. However, *simple flow aggregation* identifies other $L2$ and $L3$ clusters with size greater than r, which results in the flow errors of some hosts.

VIII. **Conclusion**

NetFlow is the traffic measurement solution most widely used by ISPs to determine the composition of the traffic mix in their networks. However, NetFlow has the problem of overrunning available memory for flow records during abnormal situations. Currently available countermeasures have their own problems. We propose an entropy based adaptive flow aggregation algorithm. This mechanism, while certainly not a panacea, provides relief from DoS attacks and other security breaches. Additionally, it significantly improves the accuracy of legitimate flows.

We choose five fields typically used to define a flow, and use 11 combinations of these five fields to define clusters. To efficiently implement the algorithm in real-time, we design a new data structure called two-dimensional hash table. Based on the concept of entropy from information theory, we use the parameter of APP to indicate the priority of clusters to be aggregated. The algorithm can efficiently identify the clusters containing attack flows as well as pick out some large normal flows belonging to the identified clusters. After identifying these clusters, the system merges flows in the clusters to metaflows, and updates information of the metaflows from new incoming flows belonging to these clusters.

We analyze the resource requirements and accuracy of our solution, and compare it with other current solutions. Experimental evaluations on synthetic and actual trace files confirm our analysis on resource requirements, and show that our solution provides better accuracy for legitimate flows. The measurements for bytes and packets are completely accurate, and measurements for flows are nearly accurate using the bitmap algorithm.

Appendix A
Complexity analysis of Algorithm 1

Assume there are N flows in an $L1$ cluster C, we give out the complexity analysis of Algorithm 1. The computation complexity depends on those operations that need to look at part or all flows in cluster C, including the operations in line 1, line 8 and line 16. There are two steps to compute the entropy of a random dimension d of cluster C. First, sort the flows by the value of dimension d such that all flows with the same value in dimension d are put together. This step can be regarded as an insertion sort with the frequency count of $N(N-1)/2$. The second step is to compute the entropy, which has the frequency count of N. Cluster C has three random dimensions, so the frequency count of the operation in line 1 is $3N(N-1)/2+3N = 3(N^2+N)/2$.

The operation in line 8 is to compute APP_f for all S_k in $L2$, $1 \le k \le n_s$, where n_s is the number of S_k in $L2$. Assume the number of flows of S_k is N_k, then $\sum_{i=1}^{n_s} N_k \le N$. S_k has two random dimensions, so the frequency count of computing its APP_f is $2N_k(N_k-1)/2+2N_k$. The frequency count of computing the APP_f of all S_k in $L2$ is $\sum_{k=1}^{n_s}(2N_k(N_k-1)/2+2N_k) = \sum_{k=1}^{n_s} N_k^2 + \sum_{k=1}^{m} N_k \le N^2+N$. So the frequency count of line 8 is $3N^2+3N$. Similarly, the frequency count of line 16 is $6N^2+6N$. From the above analysis, the computation complexity of Algorithm 1 is $O(N^2)$.

References

[1] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings of USENIX LISA*, 1999.
[2] V. Paxson, "BRO: A system for detecting network intruders in real-time," in *7th USENIX Security Symposium*, 1998.
[3] K. McCloghrie and M. Rose, "Rfc 1213," Mar. 1991.
[4] Http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml.
[5] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," in *Proc. SIGCOMM '04*, 2004.

dstIP	% of total	rejecting NetFlow byte Err.	rejecting NetFlow flow Err.	adaptive NetFlow byte Err.	adaptive NetFlow flow Err.	simple flow aggregation byte Err.	simple flow aggregation flow Err.	entropy based byte Err.	entropy based flow Err.
162.131.189.129	1.20	5.05	17.95	0.67	38.72	0	2.07	0	1.34
162.131.175.232	0.50	2.11	0.09	0.32	36.26	0	0.31	0	0
3.142.98.83	0.48	1.00	0.78	1.64	41.84	0	1.11	0	0
162.131.199.254	0.43	1.53	0.79	1.68	29.78	0	0	0	0
238.109.212.178	0.43	1.08	1.07	0.99	38.97	0	1.04	0	0
115.42.247.74	0.28	0.76	0.80	2.21	17.67	0	0	0	0
241.46.188.127	0.21	1.08	1.23	0.84	40.92	0	0.47	0	0
241.46.218.115	0.17	1.08	1.01	2.13	41.04	0	0	0	0
238.109.212.180	0.16	0.89	0.90	1.22	40.37	0	0	0	0
241.46.185.227	0.15	0.65	0.62	1.74	39.69	0	0.69	0	0

TABLE VII
Relative error (%) of dstIP breakdown on “CaidaOC48” dataset.

[6] Http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120s/120s11/12s_sanf.htm.

[7] Http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120t/120t3/netflow.htm.

[8] IPFIX Aggregation, http://www.ietf.org/internet-drafts/draft-dressler-ipfix-aggregation-02.txt.

[9] C. Estan and G. Varghese, “New directions in traffic measurement and accounting,” in *Proc. SIGCOMM ’02*, 2002.

[10] B.-Y. Choi, J. Park, and Z.-L. Zhang, “Adaptive random sampling for load change detection,” in *Proc. SIGMETRICS ’02 (extended abstract)*, 2002.

[11] N. Duffield, C. Lund, and M. Thorup, “Estimating flow distributions from sampled flow statistics,” in *Proc. SIGCOMM ’03*, 2003.

[12] N. Hohn and D. Veitch, “Inverting sampled traffic,” in *Proc. IMC ’03*, 2003.

[13] D. Plonka, “Flowscan: A network traffic flow reporting and visualization tool,” in *Proceedings of USENIX LISA*, 2000.

[14] D. Moore, K. Keys, R. Koga, E. Lagache, and kc Claffy, “Coralreef software suite as a tool for system and network administrators,” in *Proceedings of USENIX LISA*, 2001.

[15] C. Estan, S. Savage, and G. Varghese, “Automatically inferring patterns of resource consumption in network traffic,” in *Proc. SIGCOMM ’03*, 2003.

[16] K. Keys, D. Moore, and C. Estan, “A robust system for accurate real-time summaries of internet traffic,” in *Proc. SIGMETRICS ’05*, 2005.

[17] Y. Zhang, M. Roughan, C. Lund, and D. Donoho, “An information-theoretic approach to traffic matrix estimation,” in *Proc. SIGCOMM ’03*, 2003.

[18] W. Lee and D. Xiang, “Information-theoretic measures for anomaly detection,” in *Proc. IEEE Symposium on Security and Privacy*, 2001.

[19] K. Xu, Z. L. Zhang, and S. Bhattacharyya, “Profiling internet backbone traffic: Bahavior models and applications,” in *Proc. SIGCOMM ’05*, 2005.

[20] Y. Gu, A. McCallum, and D. Towsley, “Detecting anomalies in network traffic using maximum entropy estimation,” in *Proc. IMC ’05*, 2005.

[21] Y. Liu, D. Towsley, and T. Ye, “An information-theoretic approach to network monitoring and measurement,” in *Proc. IMC ’05*, 2005.

[22] CERT Coordination Center. CERT Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks, http://www.cert.org/advisories/CA-1998-01.html.

[23] CERT Coordination Center. CERT Advisory CA-2003-04 MS-SQL Server Worm, http://www.cert.org/advisories/CA-2003-04.html.

[24] CERT Coordination Center. CERT Advisory CA-2003-20 W32/Blaster worm, http://www.cert.org/advisories/CA-2003-20.html.

[25] CERT Coordination Center. CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks, http://www.cert.org/advisories/CA-1996-21.html.

[26] N. Duffield, C. Lund, and M. Thorup, “Charging from sampled network usage,” in *Proc. SIGCOMM Internet Measurement Workshop*, 2001.

[27] Http://sourceforge.net/projects/fprobe.

[28] Y. Hu, D. M. Chiu, and J. Lui, “Adaptive flow aggregation - a new solution for robust flow monitoring under security attacks,” in *Proc. NOMS ’06*, 2006.

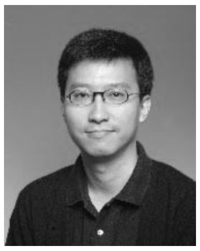
[29] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. N. Y.: John Wiley & Sons, Inc., 1991.

[30] C. Estan, G. Varghese, and M. Fisk, “Bitmap algorithms for counting active flows on high speed links,” in *Proc. IMC ’03*, 2003.

[31] Http://www.ll.mit.edu/IST/ideval/data/1998/training/.

Yan Hu received her B.E. degree in Electronic Engineering from University of Science and Technology of China and M.E. degree in Communication & Information System from Chinese Academy of Sciences, P.R. China, in 2000 and 2003, respectively. She is currently a PhD student in the Department of Information Engineering at the Chinese University of Hong Kong. Her research interests include traffic measurement and analysis, network security.

Dah-Ming Chiu (SM’02 - F’08) received a first degree from Imperial College, London, and a Ph.D degree from Harvard University. He worked in Bell-Labs, DEC and SUN before joining the Chinese University of Hong Kong in 2002. He is currently serving as the associate director of the university’s Institute of Theoretical Computer Science and Communications (ITCSC); and an associate editor of IEEE/ACM Transactions on Networking (ToN). His current research interests include network resource allocation, routing, traffic measurement and analysis, P2P networking, wireless networks and economic issues in network architecture and operations.



John C. S. Lui (M’93 - SM’02) received his Ph.D. in Computer Science from UCLA. Currently, he is the chair of the Computer Science & Engineering Department at CUHK. His research interests span both in systems as well as in theory/mathematics with the emphasis on the robustness, scalability, and security issues on the Internet. John received various departmental teaching awards and the CUHK Vice-Chancellor’s Exemplary Teaching Award, as well as the co-recipient of the Best Student Paper Awards in the IFIP WG 7.3 Performance 2005 and the IEEE/IFIP Network Operations and Management (NOMS) Conference. He is an associate editor in IEEE ToN, TPDS, TC and the Performance Evaluation Journal.