Adaptive Flow Aggregation - A New Solution for Robust Flow Monitoring under Security Attacks

Yan Hu
Dept. of Information Engineering
Chinese University of Hong Kong
Email: yhu4@ie.cuhk.edu.hk

Dah-Ming Chiu
Dept. of Information Engineering
Chinese University of Hong Kong
Email: dmchiu@ie.cuhk.edu.hk

John C.S. Lui
Dept. of CSE
Chinese University of Hong Kong
Email: cslui@cse.cuhk.edu.hk

***Abstract*— Flow-level traffic measurement is required for a wide range of applications including accounting, network planning and security management. A key design challenge is how to gracefully deal with traffic surges that exhaust the resources (memory, export bandwidth or CPU) of the flow monitor. A standard solution is to do sampling (look at one out of every n packets). This is implemented in Cisco's Netflow, a popular platform. Setting the sampling rate according to the normal traffic, however, cannot avoid overrunning available memory for flow records during abnormal situations, such as when there is a DoS attack or other security breaches. 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 a new counter-measure to deal with abnormal traffic conditions - adaptive flow aggregation. Often the reason for abnormal traffic conditions is due to security attacks. Fortunately, such attacks usually have some common patterns. For example, packets of DoS attacks have the same destination IP address, while traffic for worm spreading has the same source IP address. Our flow monitoring algorithm identifies these traffic clusters in real-time and aggregates these large amount of short flows into a few flows. Compared to currently available solutions, our solution not only alleviates the problem in memory and export bandwidth, but also guarantees the accuracy of legitimate flows. In addition, it could provide network operators with some useful information on potential security problems.

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. Especially, flow-level measurement is widely used for a wide range of applications. One example is network planning, which the ISP needs to know how the traffic load is distributed in its network. It relies on measuring the amount of traffic among pairs of customers. Flow based traffic analysis can also be used for accounting purposes, when clients are billed based on their traffic volume. Other applications include security or denial-of-service (DoS) analysis. It is also possible to see what applications are using the network by looking at traffic flows based on their port numbers.

NetFlow [1], 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 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 was created longer than the *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 (i.e., computing machines which have private processors and memory) for future analysis.

When a packet arrives at the router, NetFlow needs to look up the flow cache for an existing flow, update that entry or create a new entry. For high speed interfaces, the processor and the memory holding the flow cache can not keep up with the packet rate, so Cisco introduced sampled NetFlow [2]. There are several types of sampling methods, deterministic sampling involves random selection of one packet from the first N packets, and selection of every N^{th} packet thereafter. Random sampling selects packet randomly with a fixed sampling probability.

It is important to note 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 information; 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) The authors in [3] propose a method of adapting 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. 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 numerous 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 guarantee the accuracy of legitimate flows.
- Without aggressively exporting the records of non-terminated flows so as to make room for new ones, we would not overwhelm the collector.
- Using the information from flow aggregation, we can provide network administrators some useful information to detect DoS attacks and worm spreads.

The rest of the paper is organized as follows. We describe related work in Section II. In Section III, we describe our solutions and we provide some analysis in Section IV. Experimental evaluation based on the proposed method is presented in Section V. Conclusion is given in Section VI.

II. **Related work**

One of NetFlow's problems is the amount of data generated can be so large that it may overwhelm the collector or its network connection. Cisco's solution to this problem is to implement router-based flow aggregation [4]. Different aggregation schemes summarize NetFlow data on the router before the data is exported to the collector, resulting in lower bandwidth requirement. The IETF (Internet Engineering Task Force) working group IPFIX (Internet Protocol Flow Information eXport) also recommends aggregating similar flows into one metaflow [5]. Compared to these predefined aggregation schemes, our goal is to dynamically find flows which form a large *cluster* and aggregate these flows in real time.

In [6], Estan et al describe a method of traffic characterization that automatically groups traffic into minimal clusters of conspicuous 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. Their idea of finding the most conspicuous clusters of underlying traffic is similar to ours. The difference 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 [7], Keys et al present a system that computes multiple summaries of IP traffic in real time. They refer to sources or destinations that send or receive many packets, bytes or flows as "packet hogs", "byte hogs" or "flow hogs". This system produces these hog reports keyed by source IP, destination IP, source port and protocol, and destination port and protocol. These summaries provide information of some kind of cluster in real time, but their 12 hog reports are also predefined clusters. Another important element which is different from our solution is that their system only provides the summary information, and it does not keep any original flow information as Cisco NetFlow does.

In [8], Mahajan et al focus on network congestion caused by aggregate. They state that in both flash crowds and DoS attacks, the congestion is not due to a single flow, nor to a general increase in traffic, but due to a subset of the traffic which they call an aggregate. Their approach involves both a local mechanism for detecting and controlling an aggregate at a single router, and a cooperative pushback mechanism in which a router can ask adjacent routers to control an aggregate along its upstream path. The definition of an aggregate and the detection of aggregate in this paper is similar to ours.

The authors in [9] present algorithms that automatically identify large flows. [10] uses adaptive sampling to guarantee that the variance introduced by the variability of packet sizes does not exceed a pre-defined limit. [11] develops estimators for flow length distributions.

III. **Propose Solution**

A. Defining clusters

Our mechanism intends to protect NetFlow from running out of memory and high rate exporting 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 (and protocol), dstPort (and protocol). 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, port and IP address 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 source or destination IP address, 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 surely need to maintain a more detailed information for the accounting purpose.

Clusters are flows with the same value in some combinations of these four keys. We illustrate this using some examples. In Smurf attack [12], 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). Cluster for this type of traffic can be represented by ICMP packets to the same dstIP (the victim). The MS-SQL server worm [13] exploits a vulnerability which allows for the execution of arbitrary code on the SQL server computer due to a stack buffer overflow. Once the worm compromises a machine, it will craft packets of 376-bytes and send the 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 destination IP (sometimes with a common destination port); Packets of worm spreads often have the same source IP address (sometimes with a common destination port); Packets of port scans usually have a common destination IP address (sometimes with a common source IP address). Besides these flooding attacks, another network behavior which may cause NetFlow to run out of memory is flash crowds. It occurs when a large number of users try to access the same server simultaneously. While its intent 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 source and destination IP address as more important than the other two keys. So for defining clusters we only consider combinations which at least contain the same source or destination IP address. In other words, we would not consider a cluster which only has the same source port, and/or the same destination port. 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.

This, however, is only one of many possible ways to define and identify big clusters. Other definitions can be, for example, based on srcIP/dstIP prefixes such as 210.0.0.0/24; or based on a range of port numbers such as ports higher than 1024. Yet

combinations	**examples**
srcIP	most worms
dstIP	smurf attack ([12])
srcIP + dstIP	most portscans
srcIP + srcPort	response from syn flooding victim; response from flash crowds web server
srcIP + dstPort	W32/Blaster worm ([14])
dstIP + srcPort	N/A
dstIP + dstPort	syn flooding attacks ([15]); 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 ([13])
dstIP + srcPort + dstPort	DNS flash crowds

TABLE I

COMBINATIONS OF FOUR KEYS

other clusters can be designed based on other attributes such as Autonomous System numbers. Note that more complex definitions would require more flexible algorithms and more complicated data structures, which may impose too much overhead to real-time flow aggregation. This would be a subject of further research.

B. Data Structure

First we take fprobe [16] 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. It is an open source software distributed under GNU GPL. 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 hash to the same value. A linked list is used to store flows of this kind of hash collisions. We assume flows are defined in terms of five keys, source/destination IP address, source/destination port and protocol. When a packet arrives, the system first computes a hash value on its flow ID (five keys), and then looks it up in the hash table. It looks at every flow in the list with this hash value, to determine which flow this packet belongs to, or creates a new flow entry if the packet does not belong to any existing flow. The hash table is an appropriate data structure for flow look up, so softwares that collect network traffic and generate flow information usually use this data structure.

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 [6], it will use excessive memory, and bring too much overhead to normal flow operations like flow look up.

Our data structure is as shown in Figure 1, which is a two-dimensional hash table. One dimension of the hash table is

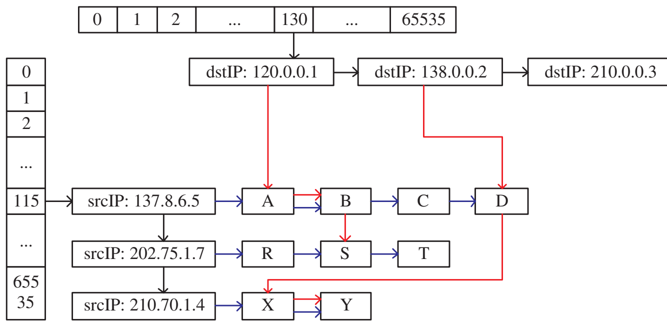

Fig. 1. data structure

the hash value based on a flow's source IP (the left table of hash number from 0 to 65535 in Figure 1), the other is the hash value based on a flow's destination IP (the top table of hash number from 0 to 65535 in Figure 1). Take source IP as an example, the hash value of a packet is computed based only on its source IP, instead of its flow ID of five keys. Packets with the same source IP will definitely be mapped to the same hash value, on the other hand, packets with a different source IP may be mapped to the same hash value because of hash collision. Each hash value node has a linked list, which consists of all source IP mapped to this hash value. For instance, in Figure 1, source IP of 137.8.6.5, 202.75.1.7 and 210.70.1.4 are all mapped to hash value 115. In addition, every source IP node has a linked list, which consists of all flows having this source IP address. The destination IP dimension of the hash table has a similar structure. The hash value of a packet is computed based on its destination IP. Each hash value node has a linked list, which consists of all destination IP addresses mapped to this hash value. For example, in Figure 1, destination IP of 120.0.0.1, 138.0.0.2, and 210.0.0.3 are all mapped to value 130. And every destination IP node has a list, which consists of all flows with this destination IP.

Every flow ID node has two parents, one is the previous node in the source IP list, the other is the previous node in the destination IP list. For example, in Figure 1, flow S has a parent of flow R in the source IP list of 202.75.1.7, and has a parent of flow B in the destination IP list of 120.0.0.1. We only consider clusters containing a fixed source or destination IP, so we compute the hash value based on these two fields. In the source/destination IP list, we put flow ID nodes sorted by destination/source IP. This data structure lets us find flows in one cluster more easily. First, all flows in one cluster of the same source or destination IP are in one list. Second, flow ID nodes in source/destination IP list are sorted by destination/source IP, so it's easy to aggregate flows in one cluster of the same srcIP plus the same dstIP.

C. Three levels of clusters

In the data structure, every IP node has a counter to indicate the number of flow nodes with this IP address. For example, in Figure 1, source IP node 137.8.6.5 has a counter of 4 to indicate there are a total of 4 flows from this source IP. With this counter, we can easily get a top list for source and destination IP address. 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 source/destination IP address, not for all combinations. In addition, different combinations have different priorities to be aggregated. For example, a combination of dstIP plus dstPort has a higher priority to be aggregated than a combination of only dstIP because it keeps more information.

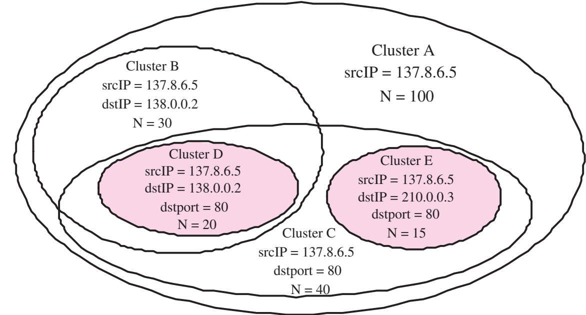

Fig. 2. three levels of clusters

Our method is to divide different clusters into three levels. There is only one global top list, so it is a mixture of source and destination IP address. Take a source IP top list node as an example, we divide different combinations with this source IP into three levels. 1) The lowest level is $L1$, flows in $L1$ cluster only have the same srcIP. 2) $L2$ cluster is about combinations of two keys, flows in $L2$ cluster can have a) the same srcIP plus destIP, b) the same srcIP plus srcPort, or c) the same srcIP plus destPort. 3) $L3$ cluster is about combinations of three keys, flows in $L3$ cluster can have a) the same srcIP plus destIP plus srcPort, b) the same srcIP plus destIP plus destPort, or c) the same srcIP plus srcPort plus dstPort.

For example, in Figure 2, the largest ellipse is a $L1$ cluster of flows with the same srcIP of 137.8.6.5 (We define it as cluster A). Flows in this $L1$ cluster also form two narrower $L2$ clusters: cluster B has the same srcIP of 137.8.6.5 plus the same dstIP of 138.0.0.2; cluster C has the same srcIP of 137.8.6.5 plus the same dstPort of 80. There are even two $L3$ clusters: cluster D and cluster E both have the same srcIP plus dstIP plus dstPort. Our definition of clusters allows clusters to overlap. If there exists a $L3$ cluster, there must be corresponding $L2$ cluster(s) and $L1$ cluster(s). Actually, $L3\ cluster \subseteq L2\ cluster \subseteq L1\ cluster$. This example has several subset relationships including: $D \subset B \subset A$, $D \subset C \subset A$, and $E \subset C \subset A$. In addition, higher level clusters have higher priority to be aggregated, because they keep more information after aggregation. In this example, when we perform aggregation, cluster D and E have the highest priority, cluster B and C have the middle priority, and cluster A has the lowest priority.

D. Algorithm for identifying clusters

Next we illustrate the algorithm to identify appropriate clusters. The objectives of this algorithm are, first, flow entries

freed during aggregating these clusters should satisfy the memory's requirement, second, the level of clusters being aggregated should be as high as possible. We first define several parameters and variables:

- P: the number of all IP nodes in the top list
- m_{max}: the memory usage that triggers aggregation
- m_{des}: the expected memory usage after aggregation
- T: the number of entries the aggregation tries to free, i.e. $T = (m_{max} - m_{des})/sizeof(a\ flow\ entry)$
- r: the smallest size of clusters the algorithm identifies
- $N_i(IP_j)$: the number of flow entries which will be freed if we aggregate all level i clusters with the IP address of node j

The algorithm identifies large clusters based on values T, r and the information in the top list. The *first step* is to compute $N_1(IP_j)$, $N_2(IP_j)$, and $N_3(IP_j)$ for every node in the top list. $N_1(IP_j)$ is the number of flow entries which will be freed if we aggregate all $L1$ clusters with the IP address of node j, so it equals to counter of node j minus 1. For example, in Figure 2, N_1 of this IP node is 99, because if we merge all flows in the $L1$ cluster - all flows with the same srcIP of 137.8.6.5 - into one flow, we can free 99 flow entries.

For a fixed source IP address, there are three kinds of $L2$ clusters. To compute N_2, we need to compute the following three values corresponding to three kinds of $L2$ clusters:

- n_{21}: number of flows which will be freed if we aggregate all *srcIP plus dstIP* clusters with this IP address
- n_{22}: number of flows which will be freed if we aggregate all *srcIP plus srcPort* clusters with this IP address
- n_{23}: number of flows which will be freed if we aggregate all *srcIP plus dstPort* clusters with this IP address

Take n_{21} as an example, we traverse flows in the list of this srcIP to find clusters with the same dstIP. n_{21} is the number of flows which will be freed if we aggregate all these srcIP plus dstIP clusters. We can get N_2 by $N_2 = \max(n_{21}, n_{22}, n_{23})$. Because flow nodes in srcIP list are sorted by dstIP, finding clusters with the same dstIP in a srcIP list only needs a counter. On the other hand, finding clusters with the same srcPort or dstPort in a srcIP list needs some temporary arrays. Computing N_3 is similar to computing N_2.

After getting $N_i(IP_j)$ ($1 \le i \le 3$, $1 \le j \le P$) for the IP nodes in the top list, the *second step* is to determine to which level we aggregate. If $\sum_j N_i(IP_j) < T \le \sum_j N_{i-1}(IP_j)$, we will aggregate to $Level\ (i-1)$. For example, if $\sum_j N_3(IP_j) < T \le \sum_j N_2(IP_j)$, then we will aggregate to $Level\ 2$. If we aggregate $L3$ clusters for all IP nodes, the sum of all N_3 is still less than T, which can not satisfy our needs. But aggregating $L2$ clusters for all IP nodes can satisfy our needs, so we choose to aggregate to $Level\ 2$. It is important to note that the level of clusters being aggregated should be as high as possible. So aggregating to $Level\ (i-1)$ means we aggregate $Level\ i$ clusters for as many IP nodes as possible, and aggregate $Level\ (i-1)$ clusters for the remaining IP nodes.

The *third step* is to determine which IP nodes to be aggregated in $Level\ i$ and which IP nodes to be aggregated in $Level\ (i-1)$. Note that $L3\ cluster \subseteq L2\ cluster \subseteq L1\ cluster$, and $N_3(IP_j) \le N_2(IP_j) \le N_1(IP_j)$. Assume aggregating to $Level\ 2$, our objective in this step is to choose as many IP nodes as possible to aggregate their $L3$ clusters. So those IP nodes whose N_2 is closer to N_3 should be chosen.

This part is implemented by Algorithm 1. First we compute $d_i(IP_j)$, which is the difference of $N_{i-1}(IP_j)$ and $N_i(IP_j)$, then sort these $d_i(IP_j)$ to $d_i(IP_{j'})$ such that $d_i(IP_{j'})$ is ascending. After that we choose the smallest $d_i(IP_{j'})$, the corresponding $N_i(IP_{j'})$ is the closest to $N_{i-1}(IP_{j'})$. If $\sum_j N_{i-1}(IP_{j'}) - d_i(IP_{j'})$ is still no smaller than T, we can choose $Level\ i$ cluster for $IP_{j'}$ and $Level\ (i-1)$ cluster for other IP nodes. Then we look at the second smallest $d_i(IP_{j'})$, and so on, until the difference is less than T. Through this algorithm, we get result t. For $\{d_i(IP_{j'}) | 1 \le j' \le t-1\}$, we choose $Level\ i$ clusters for the corresponding $IP_{j'}$, and choose $Level\ (i-1)$ clusters for other IP nodes.

Algorithm 1 clusters selection algorithm

for $j = 1$ to P
  $d_i(IP_j) = N_{i-1}(IP_j) - N_i(IP_j)$
endfor
sort $\{d_i(IP_j) | j = 1, ..., P\}$ to $\{d_i(IP_{j'}) | j' = 1, ..., P\}$,
such that $d_i(IP_{j'})$ is ascending.
$T' = \sum_j N_{i-1}(IP_j)$
for $j' = 1$ to P
  $T' = T' - d_i(IP_{j'})$
  if $T' < T$ break;
endfor
$t = j'$

One possibility is $\sum_j N_1(IP_j) < T$, then even if we aggregate all $L1$ clusters, the memory freed still can not satisfy the requirement. In this situation, the increase in number of flows is not caused by a few dominated clusters, so flow aggregation can not deal with the memory exhaustion completely.

E. Flow aggregation and export

For every node in the top list, we have decided if we should do aggregation on clusters of this IP address. We have also calculated the level of clusters and the kind of combinations (eg, srcIP plus dstIP, or srcIP plus srcPort, or srcIP plus dstPort for $L2$ clusters). After that, we find the list of flows of this IP address, merge them in selected clusters to one metaflow. Information of the metaflow comes from information of flows in this cluster. For example, if the cluster is srcIP plus dstIP, then srcIP and dstIP of the metaflow are the exact values, but its srcPort and dstPort are changed to *, denoting all possible values. Other information of this metaflow is similar to those defined in [5], number of packets/bytes is the sum of number of packets/bytes of all aggregated flows, time stamp of first seen packet (create time of the metaflow) is the minimum of this time stamp of all aggregated flows, and time stamp of last seen packet (modify time of the metaflow) is the maximum of this time stamp of all aggregated flows. The number of flows

can not be counted directly, it might be estimated using other techniques.

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 those termination criteria are met, including *inactive timer* and *active timer*. Note that criteria of observation of certain TCP flags would not be used, because these flags indicate the termination of only one flow but not the metaflow. After a metaflow is terminated and exported, flows belonging to this cluster are not aggregated to one metaflow any more. So deaggregation is done automatically based on the underlying traffic.

IV. Analysis

In this section, we analyze our algorithm (*adaptive 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*), and 4) *adaptive NetFlow* proposed in [3]. We take the implementation of fprobe as an example of NetFlow, because the implementation of Cisco NetFlow is not documented in detail.

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 Figure 1. Considering the worst case, every flow entry has different source IP and destination IP, 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.

Adaptive NetFlow may also use more memory than *basic NetFlow*. The algorithm divides the NetFlow operation into measurement bins. They do not terminate flow records during the measurement bin, but terminate all active flow records at the end of the measurement bin. A fixed size of the measurement bin is 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, while our algorithm uses a fixed amount of additional memory.

2) Export bandwidth: Besides memory, another main resource constraint is export bandwidth. Our *adaptive flow aggregation* 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: Because our algorithm intends to perform all these operations - keeping flow information, exporting flow and aggregation - in real time, it must not bring too much overhead. We will 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 source or destination IP address, then the corresponding IP node list will be so long that it would slow down flow lookup. Actually, we can define a threshold, length of IP node list reaches this threshold triggers aggregation. Another overhead to normal flow operations is that our algorithm needs to maintain a 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), and under normal conditions the number of top list entries is often less than the maximum number. So this part of overhead is not large.

We need some extra processing for performing aggregation. First, we need to traverse lists of all IP nodes in the top list to compute N_2, which is the number of flow entries that will be freed if we aggregate all $Level$ 2 clusters with this IP address. If there are $Level$ 3 clusters with this IP address, we also need to traverse this list again to get N_3. After that, we need one more traversal to do aggregation for those IP nodes which need aggregation. Assume the number of all IP nodes in the top list is P, the maximum length of IP node lists is L_m, then the running time of the aggregation operation is bounded by $3 * P * L_m$.

For finding the right sampling rate, *adaptive NetFlow* also need to maintain a histogram by performing one more addition and one subtraction for each processed packet. This histogram is the sizes of the packet counters, that is, how many flow entries have 1 packet, and how many flow entries have 2 packets, and so on. When decreasing the sampling rate, it first computes the right sampling rate using this histogram and then renormalizes all existing flow entries. While we only need to perform aggregation on flows in the lists of those IP nodes in the top list, which is a small part of the existing flow entries.

B. Accuracy

When there is an anomaly in the network, the number of flows generated would exceed the resource constraints. All kinds of countermeasures would affect the accuracy of the result. *Rejecting NetFlow* rejects all new flows when the cache is full. For *exporting NetFlow*, even if the system can process all packets and export all flow records, there are still two ways which bring inaccuracy. First, routers export NetFlow records to the collector using UDP. So flow records may be lost during periods of congestion. [17] showed the errors introduced by lost NetFlow records. Second, many post-processing analysis and visualization tools can not process this avalanche of flows. FlowScan [18] is a package used for visualizing network traffic. The author admits that its near real-time processing can not catch up when processing flows produced during most Denial-of-Service attacks. For *adaptive NetFlow*, it would automatically choose a lower sampling rate during a DoS attack, which affects accuracy of all flows. Results of our *adaptive flow aggregation* also lose information, because our solution reduces resolution for some clusters.

Comparison of lower resolution with lower sampling rate of *adaptive NetFlow* is hard to quantify. Lower sampling rates will affect the accuracy of all flows with equal probability, so inaccuracy for all kinds of aggregates (by ports, IP address, AS numbers etc.) is probabilistically equivalent. [3] presents the relative standard deviation for the number of packets and the number of bytes when estimating the traffic of any aggregate amounting to a fraction of the total traffic. On the other hand, our *adaptive flow aggregation* uses a lower resolution only for some, but not all clusters, so inaccuracy for different aggregates is quite different.

There are many analysis and visualization tools. Flowscan [18] uses NetFlow data to give detailed information about the traffic, while CoralReef [19] produces breakdowns of traffic based on packet traces instead of NetFlow data. AutoFocus [6] analyzes traffic along dynamically defined multi-dimensional clusters. These tools extract, record and help us understand the flows. They measure the traffic in the number or rate of packets, bytes and flows by breaking it down in a number of ways: by the IP protocol; by well-known services or applications; by hosts; by IP prefixes associated with networks; or by ASes and countries. These keys (protocol, applications, hosts etc.) can be predefined such as finding out how much web traffic on a link, or the top N entries such as finding hosts generating the most traffic.

We will present some examples to show the inaccuracy for different aggregates. The first example is that we aggregate $L2$ clusters of srcIP plus dstIP. Then we would get some errors if we are interested in breakdowns by the IP protocol and by applications, but we would get accurate results if we were interested in breakdowns by hosts, by IP prefixes associated with networks, or by ASes and countries. We would note that for accurate results we only mean the number of packets and bytes, because the number of flows are not counted after we merge flows in one cluster to a metaflow. Another example is that we aggregate $L2$ clusters of dstIP plus dstPort. Assume this large cluster is caused by a busy web server (produced by flash crowds) instead of a DDoS victim, such that the srcIP is meaningful. Then we get accurate results for protocol and application breakdowns. However, if the network operators are interested in the source of this web traffic, we would lose this kind of information.

From the above examples and analysis, we get the following conclusions. The inaccuracy that *adaptive flow aggregation* would bring depends on both what kind of aggregation we perform and what kind of information that network operators need. The aggregation is based on five dimensions, and the information that network operators are interested in is also about these five dimensions, because other information such as IP prefix, ASes or countries would be kept if we keep the dimension of IP address. If the dimensions that we discard during aggregation are included in the dimensions network operators are interested, then the result would be inaccurate, otherwise, it would be accurate.

In practice, keeping or discarding which dimensions is dynamically decided based on the underlying traffic. First, we choose the clusters at the highest level to keep more dimensions. Second, flow aggregation is usually triggered by a network anomaly, so the dimensions we discard are often less important, for example, large amount of spoofed source IP addresses in DoS/DDoS attacks, randomly chosen destination IP addresses in worm spreading, and randomly chosen destination ports in port scans.

C. Implementation issues

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 in real-time 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 can not 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*.

The recent rise in the use of peer-to-peer applications may also cause overload of NetFlow, because one host would open many connections to its peers and thus lead to the increase in numbers of active flows. Although unlike flows of DoS attacks and worm spreading traffic, which could be aggregated to one or a few flows, aggregating the flows originating from the same host also could mitigate the resource problem.

There are links in the network that are dominated by particular clusters, in the normal case. Network operators can use policy if they want to protect such clusters, resulting in the

	λ	τ	n	T	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
FLOW INFORMATION

algorithm looking for other clusters or perform aggregation only when they exceed their policy defined limits. Another threshold that network operators can set is r, which is the smallest number of flows in an identified cluster.

V. 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*, and our *adaptive 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. The metrics we use to evaluate these solutions are:

- memory usage - memory used at the observation point
- export bandwidth - flows exported during the past 2 minutes
- run time - time spent by the entire process
- relative error - average error for byte, packet, or flow estimates:

$$relerr = \frac{1}{n}\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\widehat{n_i} - n)^2} \qquad (1)$$

  In Equation 1, we repeat the experiment for N times, $\widehat{n}_i$ is the estimated value for number of bytes, packets or flows in the ith experiment, n is its accurate value.

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

B. Resource evaluation on synthetic trace file

We use CSIM 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 λ_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 a uniform distribution of n_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 around 3700s. Flow F is a simulated worm spread, all flows of type F have the same srcIP. It starts at 2000s, and ends at around 4000s. Flow A, B ,C, D and G are simulated normal traffic, they last during the whole duration. λ, μ and n are different for each type, so they have different characteristic, long-lived or short-lived, dense or sparse. But compared with flow E and F, their λ and μ are longer, and n is larger. Their IP address and port are randomly generated except that all flows of type G are web traffic to the same dstIP.

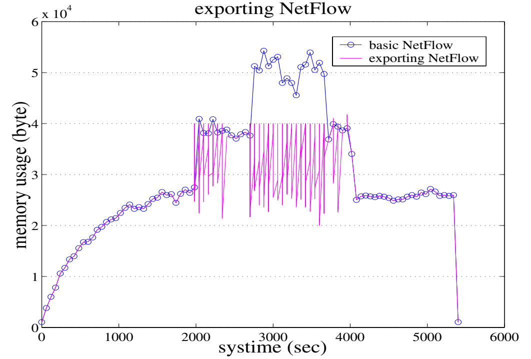

Fig. 3. memory usage for *exporting NetFlow*

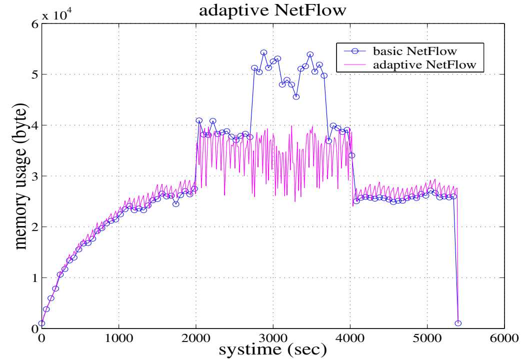

Fig. 4. memory usage for *adaptive NetFlow*

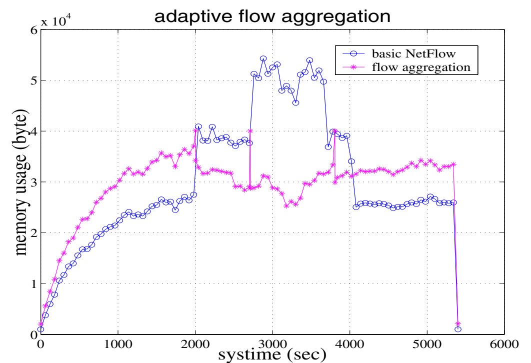

Fig. 5. memory usage for *adaptive flow aggregation*

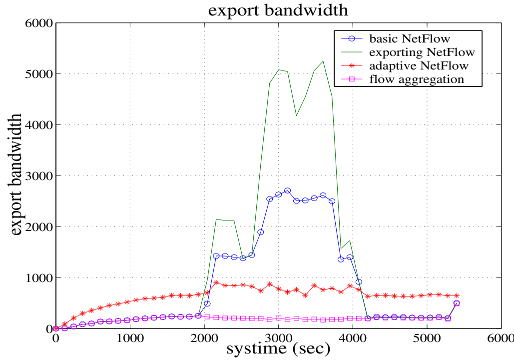

Fig. 6. export bandwidth for different solutions

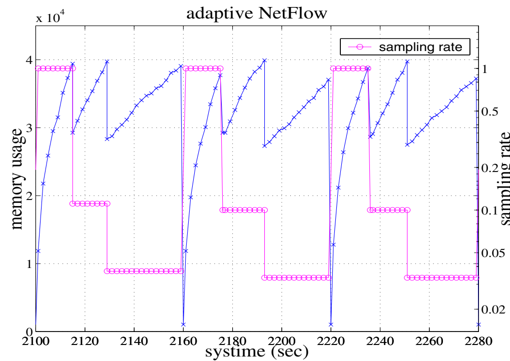

Fig. 7. memory usage and sampling rate in several measurement bins

For calculating the memory usage, we only count memory allocated for storing the active flows and record the memory usage every 10 seconds. Figure 3, Figure 4, and Figure 5 are memory usages of *exporting NetFlow*, *adaptive NetFlow*, and *adaptive flow aggregation* respectively. The solid lines with circle marker in these three figures are memory usage of *basic NetFlow*, which serves as the benchmark. We record export bandwidth every 2 minutes, which is defined as the number of flows exported during the past 2 minutes. Figure 6 is the export bandwidth for these solutions.

In these experiments, we define $m_{max} = 40000$, and $m_{des} = 30000$. When memory usage reaches m_{max}, the system performs some operations to reduce memory usage to m_{des}. In *exporting NetFlow*, the operation is to export some oldest flows. In *adaptive NetFlow*, the operation is to decrease the sampling rate as described in [3]. In *adaptive flow aggregation*, the operation is to find some large clusters and aggregate flows in these clusters. Packet processing is stopped during these operations, because input to the system is a trace file. In practice, these operations need to proceed in parallel with the processing of new packets.

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 IV-A.1. 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 Figure 7. 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 Figure 7).

For *adaptive flow aggregation*, before reaching m_{max}, its memory usage is larger than that of *basic NetFlow*, due to the overloads caused by the new data structure, as we analyzed in section IV-A.1. 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. Except flow E and F, the arrival of other flows is stable. The system aggregates all flows in type E and F to one or two flows, so export bandwidth is stable after the initial phase. The increase at the end of the observation duration is because we flush out all active flows at the end of the program.

Finally, another metric is the run time of these different processes. We find from multiple runs of these experiments that the run time of our *adaptive flow aggregation* is similar to that of *basic NetFlow*. The run time of *adaptive NetFlow* is even shorter than that of *basic NetFlow*. The reason is that other solutions check all flows in memory to look for expired flows every 2 sec (fprobe checks memory every 5 sec), while *adaptive NetFlow* only terminates all flows in memory every 1 min (we use 1 min as the size of the measurement bin).

C. Resource evaluation on "DarpaIDE" dataset

From the above section, the evaluation results on the synthetic trace file are quite consistent with what we expect. 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 [20],

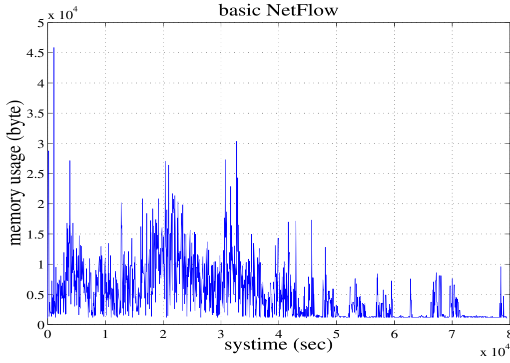

Fig. 8. memory usage of *basic NetFlow*

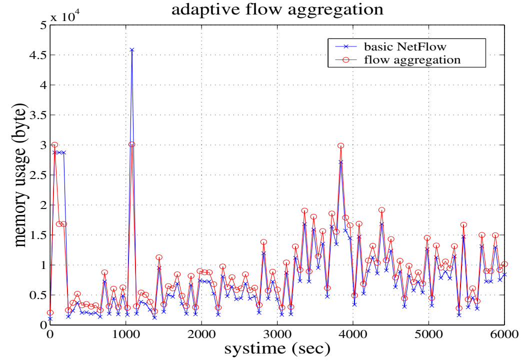

Fig. 10. memory usage of *adaptive flow aggregation*

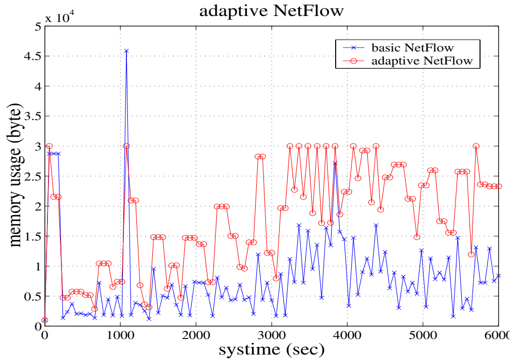

Fig. 9. memory usage of *adaptive NetFlow*

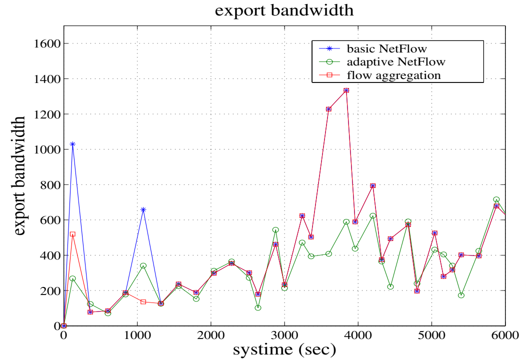

Fig. 11. export bandwidth of *adaptive NetFlow* & *adaptive flow aggregation*

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 and neptune. Figure 8 is the memory usage of *basic NetFlow*. The peak memory usage is caused by the smurf attack (ICMP packets to the same desIP). Figure 9, 10, and 11 are memory usage and export bandwidth of *adaptive NetFlow* and *adaptive flow aggregation*. In this experiment, we record memory usage every 60 sec, which is the maximum memory usage during the past 60 sec instead of memory usage at the observation point.

In this experiment, we use m_{max} = 30000, and m_{des} = 20000. From Figure 9 and 10, one can find that the memory usage of *basic NetFlow* only exceeds m_{max} at around 1000 sec. However, both *adaptive NetFlow* and *adaptive flow aggregation* may use more memory than *basic NetFlow* and their memory usage may exceed m_{max} at other points besides at around 1000 sec, so they decrease the sampling rate or perform aggregation more than once.

For *adaptive NetFlow*, before reaching m_{max}, its memory usage is often greater than that of *basic NetFlow*, even greater than that of *adaptive flow aggregation* most of the time. The reason is that many of the flows in this data set are shorter than the measurement bin of 1 min and are kept in memory longer than necessary. Its export bandwidth is similar to or less than (when decreasing the sampling rate) that of *basic NetFlow*. Figure 12 depicts its memory usage and sampling rate when the DoS attack occurred. When the smurf attack occurred at 1010 sec, the memory usage quickly reached m_{max}. To keep the memory usage bounded by m_{max}, the sampling rate was decreased once and again, with the lowest value of less than 1/100. The attack stopped at time 1046 sec, but the sampling rate would not be increased until the beginning of the next measurement bin of 1070 sec.

For *adaptive flow aggregation*, the memory usage is a little higher than that of *basic NetFlow*. Its export bandwidth is the same as or less than (when performing aggregation) that of *basic NetFlow*, as expected. Figure 13 is its memory usage when the DoS attack occurred. When its memory usage reached m_{max}, the cluster of ICMP packets to the victim was identified and flows in this cluster were merged to one metaflow. After that, the memory usage would not increase any more because all following attack packets belonged to this metaflow.

D. Accuracy evaluation on "DarpaIDE" dataset

To compare the accuracy of *adaptive NetFlow* and *adaptive flow aggregation*, we perform post-processing on the flow records exported from *adaptive NetFlow*, *adaptive 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

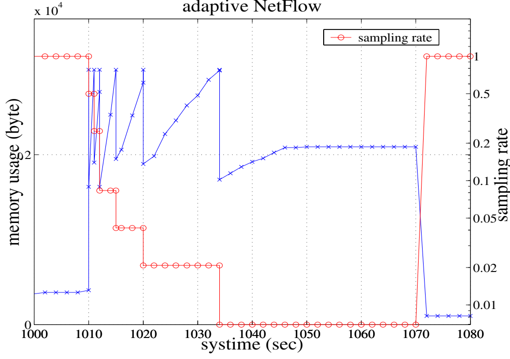

Fig. 12. memory usage of *adaptive NetFlow* under DoS attack

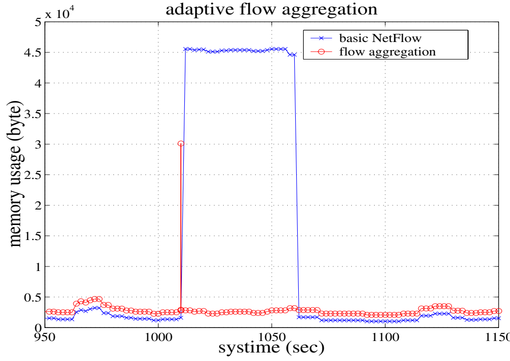

Fig. 13. memory usage of *adaptive flow aggregation* under DoS attack

bytes, packets and flows for TCP, UDP and ICMP. We repeat each experiment for 5 times, and get *relerr* using Equation 1. The configuration for these experiments is the same as in section V-C: $m_{max} = 30000$, $m_{des} = 20000$, and the size of measurement bin for *adaptive NetFlow* is 1 min. *Relerr* results for *adaptive NetFlow* and *adaptive flow aggregation* are shown in Table III.

The second post-processing step is port breakdown, which counts the number of bytes, packets and flows for different ports. For *adaptive NetFlow* and *adaptive flow aggregation*, we calculate *relerr* for the top 10 source/destination ports sorted by the number of bytes, packets and flows. For brevity, we only show *relerr* of the top 10 source ports sorted by the number of bytes of *adaptive NetFlow* and *adaptive flow aggregation* in Table IV, and omit the other five *relerr* tables.

The third post-processing step is to find the top 10 hosts by bytes, packets or flows of traffic generated/received. We get six tables similar to port breakdown. *Relerr* results of top 10 source IP addresses sorted by bytes of *adaptive NetFlow* and *adaptive flow aggregation* are shown in Table V.

From these *relerr* results, we conclude that our *adaptive flow aggregation* provides better accuracy for legitimate flows than *adaptive NetFlow*. Its measurement for the number of bytes and packets are all accurate in these scenarios, and its *relerr* for the number of flows is also lower than that of *adaptive NetFlow*. Its only *relerr* which is greater than that of *adaptive NetFlow* is flow error for ICMP, because the algorithm aggregates those ICMP flows in the smurf attack. The aggregation can keep the number of bytes and packets accurate, but can not count the number of flows directly.

adaptive NetFlow				
protocol	% of total	byte error	packet error	flow error
TCP	85.2	0.002147	0.002827	0.151821
UDP	0.6	0.009679	0.007714	0.331545
ICMP	14.2	0.212056	0.210449	0.369882
adaptive flow aggregation				
protocol	% of total	byte error	packet error	flow error
TCP	85.2	0.000000	0.000000	0.007840
UDP	0.6	0.000000	0.000000	0.059900
ICMP	14.2	0.000000	0.000000	0.663537

TABLE III

RELATIVE ERROR OF PROTOCOL BREAKDOWN

adaptive NetFlow				
srcPort	% of total	byte error	packet error	flow error
80 , tcp	66.54	0.003118	0.003229	0.166314
20 , tcp	11.45	0.002629	0.002636	0.083098
25 , tcp	0.58	0.006846	0.003628	0.029989
53 , udp	0.52	0.017324	0.012561	0.266825
21 , tcp	0.075	0.012907	0.004036	0.213127
23 , tcp	0.072	0.020535	0.012695	0.161913
123 , udp	0.069	0.029045	0.029045	0.379118
11306 , tcp	0.019	0.000000	0.000000	0.000000
11360 , tcp	0.019	0.000000	0.000000	0.000000
11304 , tcp	0.019	0.000000	0.000000	0.000000
adaptive flow aggregation				
srcPort	% of total	byte error	packet error	flow error
80 , tcp	66.54	0.000000	0.000000	0.006993
20 , tcp	11.45	0.000000	0.000000	0.000000
25 , tcp	0.58	0.000000	0.000000	0.000000
53 , udp	0.52	0.000000	0.000000	0.000000
21 , tcp	0.075	0.000000	0.000000	0.000000
23 , tcp	0.072	0.000000	0.000000	0.000000
123 , udp	0.069	0.000000	0.000000	0.000000
11306 , tcp	0.019	0.000000	0.000000	0.000000
11360 , tcp	0.019	0.000000	0.000000	0.000000
11304 , tcp	0.019	0.000000	0.000000	0.000000

TABLE IV

RELATIVE ERROR OF PORT BREAKDOWN

E. Adjusting parameters

These solutions have some parameters including, the size of the measurement bin for *adaptive NetFlow*, r for *adaptive flow aggregation* and m_{max}, m_{des} for both of them. We set $m_{max} = 30000, 35000, 40000, 45000$, and $m_{des} = m_{max} - 10000$. We choose 10s, 30s, 60s and 90s for the size of the measurement bin, and 2,4,5,10 for r. For brevity, we only give some conclusions here:

- When the size of measurement bin is small, *adaptive NetFlow* uses less memory but more export bandwidth, and the $relerr$ is low.
- The values we chose for r have little impact on memory usage, export bandwidth, and the $relerr$. This is because the sizes of all clusters identified are greater than these r.
- When m_{max} is large, $relerr$ is low.

adaptive NetFlow				
srcIP	% of total	byte error	packet error	flow error
197.218.177.69	6.16	0.004704	0.009845	0.050996
172.16.114.148	4.95	0.010789	0.012045	0.193561
208.134.241.210	3.79	0.008269	0.010362	0.166214
207.25.71.143	3.09	0.015963	0.017205	0.206470
207.25.71.29	2.79	0.039143	0.021262	0.174608
167.8.29.15	2.46	0.008376	0.008679	0.139553
207.46.130.138	2.09	0.015371	0.033316	0.267461
199.95.74.90	2.01	0.016042	0.029919	0.230012
192.168.1.10	1.25	0.008386	0.027705	0.368439
205.181.112.65	1.14	0.025910	0.022647	0.271724
adaptive flow aggregation				
srcIP	% of total	byte error	packet error	flow error
197.218.177.69	6.16	0.000000	0.000000	0.000000
172.16.114.148	4.95	0.000000	0.000000	0.033207
208.134.241.210	3.79	0.000000	0.000000	0.000000
207.25.71.143	3.09	0.000000	0.000000	0.000000
207.25.71.29	2.79	0.000000	0.000000	0.000000
167.8.29.15	2.46	0.000000	0.000000	0.000000
207.46.130.138	2.09	0.000000	0.000000	0.000000
199.95.74.90	2.01	0.000000	0.000000	0.081798
192.168.1.10	1.25	0.000000	0.000000	0.000000
205.181.112.65	1.14	0.000000	0.000000	0.000000

TABLE V

RELATIVE ERROR OF IP BREAKDOWN

VI. 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 *adaptive flow aggregation*, which identifies large clusters in real-time and aggregates large amount of short flows into a few flows. This mechanism, while certainly not a panacea, provides relief from DoS attacks and other security breaches. Additionally, it guarantees 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. One objective of the algorithm is to keep as much information as possible when performing aggregation. We divide different clusters to three levels and maintain counters to help assess their effect for aggregation. We then choose the clusters at the highest level to aggregate to minimize loss of resolution.

We analyze the resource requirement and accuracy of our solution, and compare it with other current solutions including *rejecting NetFlow*, *exporting NetFlow*, and *adaptive NetFlow*. 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.

Our future work includes: first, a formal model and analysis for accuracy comparison of *adaptive NetFlow* and our *adaptive flow aggregation*, which depends on traffic characteristics and what is the use of the flow information. In this paper, we only give some scenarios in the analysis and experimental evaluation part. Second, more experiments on additional data sets of different traffic characteristics.

Acknowledgment

This work is funded by the Area of Excellence AoE/E-01/99, and the UGC Direct Grant.

REFERENCES

[1] http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml.

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

[3] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," in *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 245–256, ACM Press, 2004.

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

[5] http://www.ietf.org/internet-drafts/draft-dressler-ipfix-aggregation-00.txt.

[6] C. Estan, S. Savage, and G. Varghese, "Automatically inferring patterns of resource consumption in network traffic," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 137–148, ACM Press, 2003.

[7] K. Keys, D. Moore, and C. Estan, "A robust system for accurate real-time summaries of internet traffic," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 85–96, 2005.

[8] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 3, pp. 62–73, 2002.

[9] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 323–336, ACM Press, 2002.

[10] B.-Y. Choi, J. Park, and Z.-L. Zhang, "Adaptive random sampling for load change detection," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 272–273, 2002.

[11] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 325–336, ACM Press, 2003.

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

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

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

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

[16] http://sourceforge.net/projects/fprobe.

[17] N. Duffield and C. Lund, "Predicting resource usage and estimation accuracy in an ip flow measurement collection infrastructure," in *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pp. 179–191, ACM Press, 2003.

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

[19] 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.

[20] http://www.ll.mit.edu/IST/ideval/data/1998/training/.