An Efficient Data Structure for Network Anomaly Detection

Jieyan Fan, Dapeng Wu, Kejie Lu, and Antonio Nucci

***Abstract*— Despite the rapid advance in networking technologies, detection of network anomalies at high-speed switches/routers is still far from maturity. To push the frontier, two major technologies need to be addressed. The first one is efficient feature-extraction algorithms/hardware that can match a line rate in the order of Gb/s; the second one is fast and effective anomaly detection schemes. In this paper, we focus on design of efficient data structure and algorithms for feature extraction. Specifically, we propose a novel data structure that extracts so-called two-directional (2D) matching features, which are shown to be effective indicators of network anomalies. Our key idea is to use a Bloom filter array to trade off a small amount of accuracy in feature extraction, for much less space and time complexity, so that our data structure can catch up with a line rate in the order of Gb/s. Different from the existing work, our data structure has the following properties: 1) *dynamic* Bloom filter, 2) combination of a *sliding window* with Bloom filter, and 3) using an insertion-removal pair to enhance Bloom filter with a *removal* operation. Our analysis and simulation demonstrate that the proposed data structure has a better space/time tradeoff than conventional algorithms. For example, for a fixed time complexity, the conventional algorithm (i.e., hash table [1]–[8]) requires a memory of 1.01G bits while our data structure requires a memory of only 62.9M bits, at the cost of losing 1% accuracy in feature extraction.**

***Index Terms*— Network security, network anomaly, edge router, bloom filter, feature extraction**

I. Introduction

With the rapid growth of Internet-based e-commerce, detection of network anomalies becomes a major concern in both industry and academia since network anomaly detection is critical to maintain availability of network services. Abnormal network behavior is usually the symptom of potential unavailability in that:

- Network anomaly is usually caused by malicious behavior, such as denial-of-service (DoS) attacks, distributed denial-of-service (DDoS) attacks, worm propagation, network scans, or email spams;
- Even if it is caused by unintentional reasons, network anomaly is often accompanied with network congestion or router failures.

Please direct all correspondence to Prof. Dapeng Wu, University of Florida, Dept. of Electrical & Computer Engineering, P.O.Box 116130, Gainesville, FL 32611, USA. Tel. (352) 392-4954, Fax (352) 392-0044, Email: wu@ece.ufl.edu, URL: http://www.wu.ece.ufl.edu.

Dr. Jieyan Fan is with the Yahoo! Inc., 701 First Ave, Sunnyvale, CA 94089.

Dr. Kejie Lu is with the Department of Electrical and Computer Engineering at the University of Puerto Rico at Mayagüez, Mayagüez, PR 00681, USA.

Dr. Antonio Nucci is with Narus, Inc., 500 Logue Avenue, Mountain View, CA 94043.

However, detecting network anomalies is not an easy task, especially at high-speed routers. One of the main difficulties arises from the fact that the data rate is too high to afford complicated data processing. An anomaly detection algorithm usually works with traffic features instead of the original traffic data itself. Traffic features can be regarded as succinct representations of the voluminous traffic, e.g., the traffic data rate is a feature of the traffic. This paper focuses on efficiently extracting the so-called 2D matching features, which are shown to be effective indicators of network anomalies (see Section II). Specifically, this paper proposes a novel data structure called Bloom filter array, to efficiently extract the 2D matching features from traffic having a data rate in the order of Gb/s.

The major contributions of this paper include

1) applying 2D matching feature to network anomaly detection;
2) introducing a counter and an insertion-removal pair vector to support counting and removal operation in a Bloom filter, which is not supported by classic Bloom filters;
3) designing a sliding window to reduce the false alarm probability caused by the boundary effect due to discrete-time sampling;
4) designing random-keyed hash functions, which provide both security and convenient extension of Bloom filter;
5) correcting an invalid assumption made in the analysis of [9] (due to technological advances in the last three decades) and re-evaluating the time complexity incurred by hash function calculations and bit comparisons;
6) analysis of time complexity, space complexity, and collision probability of Bloom filter with dynamic data set.

Features for network anomaly detection have been studied extensively in recent years. For example, Peng *et al.* [10] proposed to use the number of new source IP addresses as a feature to detect DDoS attacks, under the assumption that source addresses of IP packets observed at an edge router were relatively static in normal conditions than those during DDoS attacks. The paper further pointed out that the feature could differentiate DDoS attacks from the flash crowd, which represents the situation when many legitimate users start to access one service at the same time, e.g., when many people watch a live sports broadcast over the Internet at the same time. In both cases (i.e., DDoS attacks and the flash crowd), the traffic rate is high. But during DDoS attacks, the edge routers will observe many new source IP addresses because attackers usually spoof source IP addresses of attacking packets to hide

their identities. Therefore, the feature of the number of new source IP addresses improves those DDoS detection schemes that rely on traffic rate only. However, Peng *et al.* [10] focused on detection of DDoS attacks. It did not mention other types of network anomalies. For example, when malicious users are scanning the network, we can also observe high traffic rate but few new source IP addresses. It is very important to differentiate network scanning from flash crowd because the former is malicious but the latter is not. The 2D matching feature on different network layers, as shown by the four scenarios in Section II, can tell not only the presence of network anomalies but also their cause.

Lakhina *et al.* [11] summarized the characteristics of network anomalies under different causes. Its contribution is to help identify causes of network anomalies. For example, during DDoS attacks, we can observe high bit rate, high packet rate, and high flow rate. The source addresses are distributed over the whole IP address space. On the other hand, during network scanning, all the three rates are high, but the destination addresses, rather than the source addresses, are distributed. However, the paper did not resolve an important problem, i.e., how to extract features efficiently to match a high line rate in the order of Gb/s. This paper will address this problem.

In Ref. [12], we proposed 2D matching features to detect DDoS attacks. The features rely on the fact that most application-layer protocols generate two-way traffic between the two end-hosts. In Ref. [12], we also proposed a powerful machine learning algorithm to detect DDoS attacks, given the 2D matching features. However, the paper [12] did not focus on efficient design for feature extraction, which is the main topic of this paper.

Our analysis and simulation demonstrate that the proposed Bloom filter array has a better space/time trade-off than the conventional algorithm. For example, for a fixed time complexity, the conventional algorithm requires a memory of 1.01G bits while our Bloom filter array requires a memory of only 62.9M bits, at the cost of losing 1% accuracy in membership representation.

The rest of the paper is organized as follows. Section II defines the 2D matching features. Section III describes two basic algorithms to process the feature and points out their limitations. In Section IV, we present the proposed Bloom filter array (BFA). In Section V, we analyze the complexity of the hash table and BFA. Section VI presents simulation results to show the performance of BFA. Section VII draws conclusions.

II. 2D Matching Features

This section defines the 2D matching features, which will be used in Sections III and IV for feature extraction. This section is organized as follows. Section II-A describes the motivations of using 2D matching features. Section II-B defines the 2D matching features and lists the notations used in the paper.

A. Motivation

The motivation of using 2D matching features arises from the fact that, for most Internet applications, packets are generated from both end hosts that are engaged in communication. Information carried by packets on one direction shall match the corresponding information carried by packets on the other direction. By monitoring the degree of mismatch between the traffic flows of two directions, we can detect network anomalies. To illustrate this, let us consider the behaviors of the two-way traffic in three scenarios, namely, 1) normal conditions, 2) DDoS attacks, and 3) re-route.

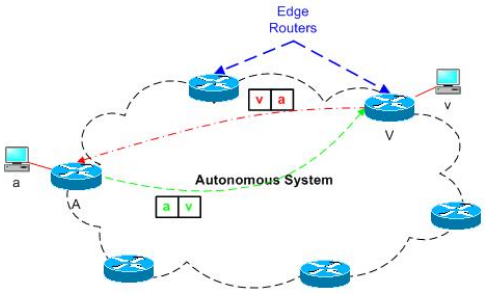


Fig. 1. Network in normal condition.

In the first scenario, when the network of an Internet service provider (ISP) works normally, information carried on both directions of communication matches, as shown in Fig. 1. Host a, a potential attacker, and host v, a potential victim, are two ends of communication (assume that host v is within the autonomous system of the ISP while host a is not). Host a sends a packet to host v and v responds a packet back to host a. Both packets pass the edge router A. From the point of view of edge router A, we define the first packet as an *inbound packet*, and the second packet as an *outbound packet*. The source address (SA) and destination address (DA) of the inbound packet match the DA and SA of the outbound packet. If the communication is based on UDP or TCP, we can further observe that the source port (SP) and destination port (DP) of the inbound packet match the DP and SP of the outbound packet. Therefore, edge routers of the autonomous system can observe matched inbound and outbound packets in normal conditions. In the example of Fig. 1, it is assumed that the border gateway protocol (BGP) routing makes the inbound packets and the corresponding outbound packets pass through the same edge router. If the BGP routing makes the inbound packets and the corresponding outbound packets go through different edge routers as shown in Fig. 3, the matching can still be achieved by a global analyzer proposed by Lu *et al.* [12], i.e., multiple edge routers in an autonomous system convey the unmatched inbound packets and the corresponding outbound packets to a centralized matcher (global analyzer), which has the routing information of the whole autonomous system.

In the second scenario, when attackers launch spoofed-source-IP-address DDoS attacks [13], the edge routers can observe many unmatched inbound packets, as shown in Fig. 2. Since source addresses of inbound packets are spoofed, the outbound packets are routed to the nominal destinations, i.e., b and c in Fig. 2, which do not pass through edge router A any more. In this case, edge router A will observe many unmatched inbound packets.

In the third scenario, as shown in Fig. 3, the number

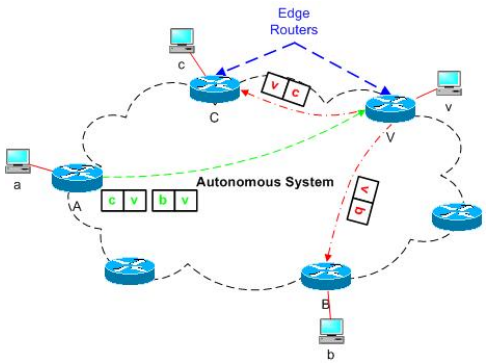


Fig. 2. Source-address-spoofed packets.

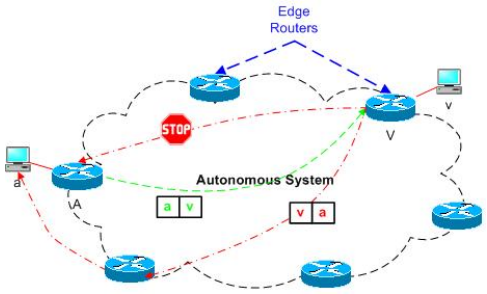


Fig. 3. Reroute.

of unmatched inbound packets observed by edge router A is increased due to a failure of the original route and re-route of outbound packets to another edge router. A global analyzer proposed by Lu *et al.* [12] can address this problem as mentioned in the first scenario.

All the above scenarios seem to suggest that the number of *unmatched inbound packets* observed by an edge router is a good feature for network anomaly detection. However, usually, this is not true because traffic volume from one end to the other is not symmetric, typically. In Fig. 1, if host a is a client uploading a large file using the File Transfer Protocol (FTP) [14] to host v, there will be much more packets from a to v than those from v to a. Uploading file to an FTP server is a normal behavior but the number of unmatched inbound packets is very high in this case.

Therefore, it is more appropriate to use flow-level quantities (instead of packet-level quantities) as features for network anomaly detection. As in the above FTP case, when a TCP connection is established, all packets on one direction constitute one flow and packets on the reverse direction constitute another flow. No matter how many packets are sent on each direction, there are only one inbound flow and only one outbound flow. They match in IP addresses and port numbers. Therefore, we call the number of *unmatched inbound flows* as a two-directional (2D) matching feature.

At a first glance, it seems like 2D matching feature only makes sense to TCP, as UDP does not necessarily generate 2D traffic. In practice, however, although UDP is used at the transport layer, application layer protocols generally work in an interactive way, such as SNMP and TFTP. The dominance of application protocols generating 2D traffic makes the 2D matching feature significant.

TABLE I
NOTATIONS AND DEFINITIONS FOR 2D MATCHING FEATURES

Γ	: (Discrete-time) sampling interval.
t_i	: The ith sampling time epoch, where $t_{i+1} = t_i + \Gamma$ and $i \in \mathbb{Z}_+$.
p	: An inbound packet.
$P(t_i)$	: The set of all inbound packets arriving during $[t_i, t_{i+1})$, i.e., $\{p : p$ arrives during $[t_i, t_{i+1})\}$
p'	: An outbound packet.
$P'(t_i)$	: The set of all outbound packets arriving during $[t_i, t_{i+1})$, i.e., $\{p' : p'$ arrives during $[t_i, t_{i+1})\}$
$f(p)$	: Inbound signature of p.
$f'(p')$	: Outbound signature of p'.
$X(t_i)$	: The set of signatures of all inbound packets during $[t_i, t_{i+1})$, i.e., $\{f(p) \vert p \in P_i\}$
$Y(t_i)$	: The set of signatures of all outbound packets during $[t_i, t_{i+1})$, i.e., $\{f'(p') \vert p' \in P'_i\}$
$D(t_i)$	: $X(t_i) - Y(t_i)$
$\vert D(t_i)\vert$	: The number of *UIF* (i.e., feature of interest).

2D matching features are shown to be effective indicators of network anomalies [12].[1] However, extraction of 2D matching features at high-speed edge routers is not an easy task. We will address this issue in Sections III and IV.

B. Definition of 2D Matching Features

We first define three terms.

Definition 1: Signature is the information of interest, carried in traffic.

The exact definition of signature depends on the specific application targeted. For example, to detect SYN flood DDoS attacks, as we only care about TCP SYN packet and SYN-ACK packet, we may use a 5-tuple signature <SA, SP, DA, DP, sequence number> for inbound packets and <DA, DP, SA, SP, ACK number – 1> for outbound packets. Generally, we can use the first 4 tuples as the signature for TCP or UDP packets.

Definition 2: A *flow* is a set of the packets with the same signature and the same direction.

For example, a TCP connection between two ends generates two flows with different directions.

Definition 3: An *unmatched inbound flow* (UIF) is an inbound flow that has no corresponding outbound packet arriving at an intended edge router within a time period Γ.

Note that we use a time constraint Γ in the definition of UIF because it takes time for an outbound packet to arrive. The value of Γ depends on the round trip time (RTT) of the connection.

Table I lists the notations used in this paper, where $\mathbb{Z}_+$ represents the nonnegative integer set. $|D(t_i)|$ represents the 2D matching feature, i.e., the number of UIF, and it is sampled at discrete time t_i. In the following sections, we present algorithms to extract $|D(t_i)|$ from the traffic at edge routers.

III. BASIC ALGORITHMS

This section presents two basic algorithms to process and store the 2D matching features, namely, the Hash Table Algorithm and Bloom filter.

[1] 2D matching features are good indicators of DDoS attacks with spoofed source IP addresses but are not good indicators of DDoS attacks with non-spoofed source IP addresses.

A. Hash Table Algorithm

From the discussion in Section II-B, we know the general procedure to extract $|D(t_i)|$ from traffic is:

1) When a p comes, if there is no entry for $f(p)$ in the buffer, create one entry for $f(p)$ and set the state of that entry to "UNMATCHED";
2) When a p' comes, if there is an entry for $f'(p')$ in the buffer, set the state of that entry to "MATCHED";
3) At time t_{i+1}, assign the number of entries with state "UNMATCHED" to $|D(t_i)|$.

So typically we need three operations: insertion, search and removal[2].

A basic algorithm to do this is to use a hash table. Suppose the signature extracted from a packet is b bits long. We organize the buffer into a table with l cells of $b+1$ bits each. The extra one bit is the state bit. We also have K hash functions h_i:$S \mapsto \mathbb{Z}_l$, where $i \in \mathbb{Z}_K = \{0, 1, \ldots, K-1\}$, and S is the data set of interest, e.g., signature domain.

The insertion operation is as follows:

1) Let $i = 0$.
2) $k = h_i(f(p))$.
3) If the kth cell is empty, insert $f(p)$ into this cell, set its state bit to 0 (i.e., "UNMATCHED"), and then exit the loop.
4) If the kth cell holds $f(p)$, exit the loop.
5) Otherwise, collision happens, let $i = i + 1$, and repeat steps 2 to 4 until $i = K$. If $i = K$ and there is still a collision, an error report is generated and the insertion operation terminates.

The search operation is similar to the insertion operation. The difference is the return of the operation:

1) At step 3, if an empty cell is found, search operation returns *false*.
2) At step 4, if a cell storing $f(p)$ is found, search operation returns *true*.

The removal operation simply sets the state bit to 1 (i.e., "MATCHED") if a cell holding $f(p)$ is found during the search operation.

B. Bloom Filter

The hash table algorithm can be used for offline traffic analysis or analysis of low data-rate traffic but it cannot catch up with a high data rate at edge routers. To address this limitation, one can use Bloom filter [9]. Compared to the hash table algorithm, Bloom filter reduces space/time complexity by allowing small degree of inaccuracy in membership representation, i.e., an $f(p)$, which does not appear before, may be falsely identified as present.

Bloom filter stores data in a vector V of M elements, each of which consists of one bit. Bloom filter also uses K hash functions h_i:$S \mapsto \mathbb{Z}_M$, where $i \in \mathbb{Z}_K$. Fig. 4 describes the insertion and search operations of Bloom filter.

Although Bloom filter has better performance in the sense of space/time trade-off, it cannot be directly applied to our application because of the following problems:

[2]Setting the state to "MATCHED" is actually the removal operation.

```
1)  function BloomFilterInsert(V, s)
2)      for ∀i ∈ Z_K do
3)          V[h_i(s)] ← 1
4)  end function
5)  function BloomFilterSearch(V, s)
6)      for ∀i ∈ Z_K do
7)          if V[h_i(s)] ≠ 1 then
8)              return false
9)      end for
10)     return true
11) end function
```

Fig. 4. Bloom Filter Operations

1) Bloom filter does not provide removal functionality. Since one bit in the vector may be mapped by more than one item, it is unsuitable to remove the item by setting all bits indexed by its hash results to 0.
2) Bloom filter does not have counting functionality. Although the counting Bloom filter [15] can be used for counting, it replaces a bit with a counter, which significantly increases the space complexity.
3) Sampling 2D matching features in discrete time results in boundary effect, as shown in Fig. 5(a). A p arrives at time t'_1 and its matched packet p' arrives at time t'_2. Since $f(p)$ is counted in X_i whereas $f'(p')$ is not counted in Y_i, p is counted as an unmatched inbound packet even though $t'_2 - t'_1 < \Gamma$. Therefore, boundary effect increases the false alarm rate.
4) In Fig. 5(a), we did not consider the scenario that a p' may arrive before its matched packet p, as shown in Fig. 5(b). When a p' arrives at time t'_1, $f'(p')$ is not in the buffer, so we do nothing. At time t'_2, its matched packet p arrives and $f(p)$ will be placed in the buffer. When we sample $|D(t_i)|$ at time t_{i+1}, p is regarded as an unmatched inbound packet. This early-arrival problem also increases the false alarm rate.

Next, we propose a Bloom filter array to address the above problems.

IV. Bloom Filter Array

The good space/time trade-off motivates us to apply Bloom filter to 2D feature extraction. But we need to address the problems of Bloom filter mentioned in Section III-B. Our idea is to design a Bloom filter array (BFA) with the following functionalities, not available in the original Bloom filter [9] and [16]:

1) *Removal functionality*: We implement insertion and removal operations synergistically by using insertion-removal pair vectors. The trick is that, rather than removing $f'(p')$ from the insertion vector, we create a removal vector and insert $f'(p')$ into the removal vector.
2) *Counting functionality*: We implement this by introducing counters in Bloom filter array. The value of a counter is changed based on the query result from an insertion/removal operation.
3) *Boundary effect abatement*: We use multiple time slots and a sliding window to mitigate the boundary effect.
4) *Resolving the early-arrival problem*: which is achieved by storing information of not only inbound packets but

(a) Boundary problem

(b) A p' arrives before a matched p with $\bar{t}_2 - \bar{t}_1 < \Gamma$

Fig. 5. Scenarios of the problems caused by Bloom filter

also outbound packets. In this way, when a p arrives and the signature of its matched p' is present, we do not count p in $|D(t_i)|$.

The rest of the section is organized as follows. In Sections IV-A and IV-B, we present the data structure and algorithm of BFA, respectively. We describe the sliding window and random-keyed hash functions used in BFA in Sections IV-C and IV-D, respectively.

A. Data Structure

To address the boundary effect, we partition a discrete-time interval of Γ into w time slots, where w is the number of slots enough to mitigate the boundary effect (see Section IV-C). Assume the length of a slot is γ. Then, we have $\Gamma = w \times \gamma$. The data structure of BFA is as follows:

- An array of bit vectors $\{IV_j\}$ ($j \in \mathbb{Z}_+$), where IV_j is the jth insertion vector holding $f(p)$ in slot $[\tau_j, \tau_{j+1})$, where $\tau_{j+1} = \tau_j + \gamma$.
- An array of bit vectors $\{RV_j\}$ ($j \in \mathbb{Z}_+$), where RV_j is the jth removal vector holding $f'(p')$ in slot $[\tau_j, \tau_{j+1})$.
- An array of counters $\{C_j\}$ ($j \in \mathbb{Z}_+$), where C_j is used to count the number of UIF in slot $[\tau_j, \tau_{j+1})$.

Since the two-way flows need to be matched within a time interval of length Γ, we only need to keep information within a time window of length Γ. That is, if the current slot is $[\tau_j, \tau_{j+1})$, only $\{IV_{j-w+1}, \ldots, IV_j\}$, $\{RV_{j-w+1}, \ldots, RV_j\}$, and $\{C_{j-w+1}, \ldots, C_j\}$ are kept in memory.

B. Algorithm

As shown in Fig. 6, our algorithm for BFA consists of three functions, namely, *ProcInbound*, *ProcOutbound* and *Sample*, which are described as below.

Function *ProcInbound* is to process inbound packets. It works as below. When a p arrives during $[\tau_j, \tau_{j+1})$, we increase C_j by 1 and insert $f(p)$ into IV_j if none of the following conditions is satisfied:

1) $f(p)$ is stored in at least one $RV_{j'}$, where $j - w + 1 \leq j' \leq j$;
2) $f(p)$ is stored in IV_j.

Condition 1 being true means that the outbound flow of p has been observed previously; so we should not count p as a packet of an UIF. Condition 2 being true means that the inbound flow, to which p belongs, has been observed in the current slot j; so we should not count the same inbound flow again. In Function *ProcInbound*, a and b are two flags with initial value **false**. Flag a is used to indicate condition 1 (line 3 to 4) and flag b for condition 2 (line 5 to 6). If they are both

```
1)  function ProcInbound(p)
2)      a ← false, b ← false
3)      if ∃j', j−w+1 ≤ j' ≤ j, such that BloomFilterSearch(RV_j', f(p))
        returns true then
4)          a ← true
5)      if BloomFilterSearch(IV_j, f(p)) returns true then
6)          b ← true
7)      if a and b are both false
8)          C_j ← C_j + 1
9)          BloomFilterInsert(IV_j, f(p))
10)     end if
11) end function
12) function ProcOutbound(p')
13)     for j' ← j to j − w + 1
14)         if BloomFilterSearch(RV_j', f'(p')) returns true
15)             break
16)         if BloomFilterSearch(IV_j', f'(p')) returns true
17)             C_j' ← C_j' − 1
18)     end for
19)     BloomFilterInsert(RV_j, f'(p'))
20) end function
21) function Sample(j)
22)     return C_{j−w+1}
23) end function
```

Fig. 6. Bloom Filter Array Algorithm

false, we increase C_j by one to indicate a new potential UIF (line 7 to 10).

Function *ProcOutbound* is to process outbound packets. It works as below. When a p' arrives during $[\tau_j, \tau_{j+1})$, we check whether we need to update counter $C_{j'}$ for each j' ($j-w+1 \leq j' \leq j$). Specifically, for each j' ($j-w+1 \leq j' \leq j$), decrease $C_{j'}$ by one if both of the following conditions are satisfied:

1) $f'(p')$ is not contained in $RV_{j'}$;
2) $f'(p')$ is contained in $IV_{j'}$.

Condition 1 being true means that no packet from the outbound flow of p' arrives during the j'th time slot. Condition 2 being true means that the inbound flow of p' has been observed in the j'th slot. Satisfying both conditions means that the inbound flow of p' has been counted as a potential UIF; hence, upon the arrival of p', the inbound flow of p' is matched and we need to decrease $C_{j'}$ by one. In Function *ProcOutbound*, Line 13 starts a loop to iterate j' from j to $j-w+1$. Condition 1 is checked in lines 14 to 15 and Condition 2 is checked in lines 16 to 17. Note that the loop exits (line 15) if $RV_{j'}$ contains $f'(p')$; this is because an outbound packet of the same *flow* arrived in that j'th slot and hence the buffer of the $\bar{j}$th slot (for each $\bar{j} < j'$) has already been checked.

Function *Sample* is to extract the 2D matching feature. When we execute Function *Sample* at the end of the jth slot (i.e., at time τ_{j+1}), the output is $|D(\tau_{j-w+1})|$ instead of $|D(\tau_j)|$ since a time lag of Γ (w slots) is needed for 2D matching.

C. Round Robin Sliding Window

The algorithm presented in Section IV-B has a drawback in memory allocation. Specifically, at epoch τ_{j+1}, we sample $|D(\tau_{j-w+1})|$, and then we need to throw away the buffer for the $(j-w+1)$th slot, and create a new buffer for the $(j+1)$th slot. This is inefficient for most operating systems. A better memory allocation strategy is to use the useless buffer of the $(j-w+1)$th slot for the new $(j+1)$th slot, saving the cost of deleting the existing buffer and acquiring a new buffer. This is the idea of our round-robin sliding window.

Our new memory allocation scheme is the following. We allocate a memory area of fixed size for w insertion vectors $\{IV_j\}$, w removal vectors, $\{RV_j\}$, and w counters $\{C_j\}$, where $j \in \mathbb{Z}_w$. The insertion vector, removal vector, and counter for the jth slot are $IV_{j\%w}$, $RV_{j\%w}$, and $C_{j\%w}$, respectively. Here, $\%$ stands for modulo operation. We also define a pointer I to point to the current slot. Then, rather than deleting a useless buffer and acquiring a new buffer for the new slot, we simply update the pointer by $I = (I+1)\%w$. Fig. 7 shows the improved version of BFA, based on the round-robin sliding window.

```
1)  function ProcInbound(p)
2)      a ← false, b ← false
3)      if ∃j', j' ∈ {(I − w + 1)%w, (I − w + 2)%w, ..., I%w}, such
        that BloomFilterSearch(RV_j', f(p)) returns true then
4)          a ← true
5)      if BloomFilterSearch(IV_I, f(p)) returns true then
6)          b ← true
7)      if a and b are both false then
8)          C_I ← C_I + 1
9)          BloomFilterInsert(IV_I, f(p))
10)     end if
11) end function
12) function ProcOutbound(p')
13)     for j' ← I to (I − w + 1)%w
14)         if BloomFilterSearch(RV_j', f'(p')) returns true then
15)             break
16)         if BloomFilterSearch(IV_j', f'(p')) returns true then
17)             C_j' ← C_j' − 1
18)     end for
19)     BloomFilterInsert(RV_I, f'(p'))
20) end function
21) function Sample()
22)     I ← (I + 1)%w
23)     return C_I
24) end function
```

Fig. 7. Bloom Filter Array Algorithm using sliding window

D. Random-Keyed Hash Functions

In previous sections, we assume K hash functions are given *a priori*. However, choosing hash functions appropriately is not trivial due to the following two concerns.

First, K is a user-specified parameter, subject to change. But for a value of K that a user[3] chooses, it is not desirable to require the user to manually select K hash functions from a large pool of hash functions provided by the manufacturer. Also, it wastes memory to store a large pool of hash functions.

Second, to improve security, the K hash functions need to be changed over time. Otherwise, if an attacker knows the hash functions, he can generate such attack packets that for any two packets p and q, $f(p) \neq f(q)$ but $h_i(f(p)) = h_i(f(q))$, $i \in \mathbb{Z}_K$. The consequence is that even if there are many attack packets with different signatures, the BFA algorithm will regard them as belonging to the same flow. So, the number of UIF for these packets is only one. This causes security vulnerability.

We address the aforementioned two problems by using keyed hash functions, i.e., we only need one kernel hash function and K randomly generated keys. Specifically, the ith hash function $h_i(x)$ is simply $h(key_i, x)$, where h is a predefined kernel hash function and $\{key_i\}$ ($i \in \mathbb{Z}_K$) are randomly generated keys. For example, we can use MD5 Digest Algorithm [17] as the hash function. Since MD5 takes any number of bits as input, we can organize key_i and x into a bit vector and apply MD5 to it.

Using keyed hash functions, the first concern (varying K) can be addressed straightforwardly. Specifically, when K is changed, we simply generate a corresponding number of random keys. Applying these K keys to the same kernel hash function, we obtain K hash functions. Hence, our method has two advantages: 1) the number of hash functions can be specified on the fly; 2) hash functions are determined on the fly, instead of being stored *a priori*, resulting in storage saving.

The second concern (changing hash functions) can also be addressed if the keys are periodically changed. Even if the kernel hash function is disclosed, it is still very difficult, if not impossible, for an attacker to guess the changing random keys.

Note that the collision probability of the hash functions is not affected due to the use of keyed hash functions. In the case of random-keyed hash functions, the collision probability of $h_i(x)$ depends on not only the collision probability of h but also the correlation between key_i and x. Since random number generator techniques are so mature that we can assume independence between key_i and x, introduction of random keys has no effect on the collision probability.

V. Complexity Analysis

This section compares the hash table, Bloom filter, and our BFA. The section is organized as follows. In Section V-A, we analyze the space/time trade-off for the three algorithms. Section V-B addresses how to optimally choose parameters of BFA.

A. Space/Time Trade-off

Space/time trade-off for both Hash Table and Bloom filter was analyzed by Bloom [9]. However, the analysis in [9] is not directly applicable to our setting due to the following reasons:

1) A static data set was assumed by Bloom [9]. However, our feature extraction deals with a dynamic data set, i.e., the number of elements in the data set changes over time. Hence, new analysis for a dynamic data set is needed. In addition, Bloom [9] only considered the search operation due to the assumption of static data sets. Our feature extraction, on the other hand, requires

[3]A user here is a network operator who wants to use our BFA and detection technique to detect network anomalies.

TABLE II
NOTATIONS FOR ANALYSIS

N	: Random variable representing the number of different flows recorded.
ϕ	: Empty ratio.
η	: Collision probability, i.e., the probability that an item is falsely identified to be in the buffer.
R	: Flow arrival rate, which is assumed to be constant.

three operations, i.e., insertion, search, and removal, for dynamic data sets.

2) Bloom [9] assumed bit-comparison hardware in time complexity analysis. However, current computers usually use word (or multiple-bit) comparison, which is more efficient than bit-comparison hardware. Hence, it is necessary to analyze the complexity based on word comparison.
3) The time complexity obtained by Bloom [9] did not include hash function calculations. However, hash function calculation dominates the overall time complexity, e.g., calculating one hash function based on MD5 takes 64 clock cycles [18], while one word-comparison usually takes less than 8 clock cycles [19].

For the above reasons, we develop new analysis for the hash table and Bloom filter in Sections V-A.1 and V-A.2, respectively. Section V-A.3 provides the analysis for BFA while Section V-A.4 shows numerical results to compare the performance of the three algorithms. Table II lists the notations used in the analysis.

1) Analysis for Hash Table: Denote by M_h the size of a hash table in bits (i.e., space complexity) and by T_h the random variable representing the number of hash function calculations for an unsuccessful search (i.e., time complexity).

Let us consider search operation first. Upon the arrival of packet p, the search algorithm checks if $f(p)$ is in the table. Because an unsuccessful search will continue the loop until an empty cell is found, it consumes more time than a successful one does. In addition, it is very difficult to analyze the time complexity of a successful search since the complexity depends on the distribution of flow signatures and the data rate of each flow. For this reason, we only consider the time consumed for an unsuccessful search, which is a conservative estimate of the average time complexity of a search. Recall that, as mentioned in Section III-A, the hash table has l cells of $b+1$ bits each, such that $M_h = l(b+1)$. Given the condition that N flows have been recorded by the hash table, the empty ratio is

$$\phi = \frac{l-N}{l} = \frac{M_h - N(b+1)}{M_h}. \tag{1}$$

In each loop, the search algorithm calculates one hash function and checks the addressed entry. If the entry is not empty, next loop is executed. The conditional probability that the loop is executed for x times for a given n follows a geometric distribution as below

$$\Pr[T_h = x | N = n] = \phi(1-\phi)^{x-1}. \tag{2}$$

Therefore the conditional expectation of T_h is

$$\begin{aligned} E[T_h|N=n] &= \sum_{x=1}^{\infty} x\phi(1-\phi)^{x-1} \\ &= \frac{1}{\phi} \\ &= \frac{M_h}{M_h - n(b+1)}. \end{aligned} \tag{3}$$

Since the table records data for the duration of Γ, the maximum number of different flows that we need to store in the buffer is $R\Gamma$. Then the expectation of T_h is

$$E[T_h] = \sum_{n=0}^{R\Gamma} \Pr[N=n] E[T_h|N=n]. \tag{4}$$

Assume N has a uniform distribution

$$\Pr[N=n] = \frac{1}{R\Gamma+1}. \tag{5}$$

Applying Eq. (5) to Eq. (4), we obtain the expectation of T_h

$$E[T_h] = \frac{1}{R\Gamma+1} \sum_{n=0}^{R\Gamma} \frac{M_h}{M_h - n(b+1)}. \tag{6}$$

Since the time to insert $f(p)$ into or remove $f(p)$ from a given entry is much shorter than that to find the proper entry, the time complexities of insertion and removal operations are almost the same as that of the search operation. Eq. (6) gives the space/time trade-off (i.e., M_h vs. T_h) of the hash table method.

2) Analysis for Bloom Filter: First of all, we consider the space complexity of Bloom filter. Denote by M_b the length of the vector V used by Bloom filter (see Section III-B). The choice of M_b will affect the accuracy of the search function, *BloomFilterSearch* (see Fig. 4). The reason is the following.

When signatures of N flows are stored in V, ϕ, denoting the percentage of entries of V with value 0, is

$$\phi = \left(1 - \frac{K}{M_b}\right)^N, \tag{7}$$

where K is the number of hash functions. Assuming $K \ll M_b$, as is certainly the case, we can approximate ϕ as

$$\phi \approx \exp\left(-\frac{KN}{M_b}\right). \tag{8}$$

Function *BloomFilterSearch*(V, s) falsely identifies s to be stored in V if and only if results of all K hash functions point to bits with value 1, which is known as a collision. Denote by η_N the collision probability under the condition that N flows have been recorded. Then

$$\eta_N = (1-\phi)^K = \left[1 - \exp\left(-\frac{KN}{M_b}\right)\right]^K. \tag{9}$$

Therefore, the average collision probability is

$$\eta = \sum_{n=0}^{R\Gamma} \eta_n \Pr[N=n] = \frac{1}{R\Gamma+1} \sum_{n=0}^{R\Gamma} \left[1 - \exp\left(-\frac{Kn}{M_b}\right)\right]^K, \tag{10}$$

where N is assumed to be uniformly distributed as in Eq. (5). From Eq. (10), it can be observed that η decreases with M_b if K is fixed. Based on Eq. (10), we can denote M_b as a function of η and K as below

$$M_b = \alpha_{R\Gamma}(\eta, K). \tag{11}$$

Eq. (11) gives the space complexity of Bloom filter as a function of collision probability and the number of hash functions.

Now, let us consider the time complexity of Bloom filter. Denote by T_b the random variable representing the number of hash function calculations.

Function *BloomFilterInsert* always calculates all the K hash functions, that is,

$$T_b|\{\textit{BloomFilterInsert} \text{ is executed}\} \equiv K, \tag{12}$$

where "|" followed by an event means a condition and "$\equiv$" means equality with probability 1.

For function *BloomFilterSearch*, we first consider a special case that *BloomFilterSearch* returns true. In this case, all K hash functions need to be calculated. So

$$T_b|\{\textit{BloomFilterSearch} \text{ returns true}\} \equiv K. \tag{13}$$

This fact will be used in the analysis for BFA (see Section V-A.3).

In general,

$$\begin{aligned} &Pr[T_b = x|\text{N=n and } \textit{BloomFilterSearch} \text{ is executed}] \\ =&\begin{cases} \phi(1-\phi)^{x-1} & x < K \\ (1-\phi)^{K-1} & x = K \end{cases}. \end{aligned} \tag{14}$$

Hence, the conditional expectation of T_b is

$$\begin{aligned} &E[T_b|\text{N=n and } \textit{BloomFilterSearch} \text{ is executed}] \\ =&\sum_{x=1}^{K-1} x\phi(1-\phi)^{x-1} + K(1-\phi)^{K-1} \\ =&\frac{1-\left[1-\exp\left(-\frac{Kn}{\alpha_{R\Gamma}(\eta,K)}\right)\right]^K}{\exp\left(-\frac{Kn}{\alpha_{R\Gamma}(\eta,K)}\right)} \\ \overset{denote}{=}&\beta_n(\eta, K). \end{aligned} \tag{15}$$

Averaging over N at both sides of Eq. (15), we get the expectation of T_b under the condition that *BloomFilterSearch* is executed, i.e.,

$$\begin{aligned} &E[T_b|\textit{BloomFilterSearch} \text{ is executed}] \\ =&\frac{1}{R\Gamma+1}\sum_{n=0}^{R\Gamma}\beta_n(\eta, K). \end{aligned} \tag{16}$$

If we know the two prior probabilities, i.e., the probability that *BloomFilterSearch* is executed, denoted by P_s, and the probability that *BloomFilterInsert* is executed, denoted by P_i, then we can get

$$E[T_b] = \frac{P_s}{R\Gamma+1}\sum_{n=0}^{R\Gamma}\beta_n(\eta, K) + P_i K. \tag{17}$$

Eq. (17) gives the time complexity of Bloom filter in terms of number of hash function calculations.

3) Analysis for BFA: Once again, we analyze the space complexity of BFA first. The techniques in Section V-A.2 can be applied here since BFA is originated from standard Bloom filter. However, there are some differences between these two schemes. As described in Section IV, BFA has multiple buffers such as IV_j, RV_j, and C_j, $j \in \mathbb{Z}_w$. Therefore, the storage size for BFA, denoted by M_a (in bits), is $w(2 \times M_v + L)$, where M_v is the size of each insertion or removal vector, and L is the size of each counter in bits.

Similar to Eq. (10), the collision probability is

$$\eta = \frac{1}{R\gamma+1}\sum_{n=0}^{R\gamma}\left[1-\exp\left(-\frac{Kn}{M_v}\right)\right]^K. \tag{18}$$

Note that length of each time slot of BFA is γ, so that the upper limit of the summation operator is $R\gamma$ rather than $R\Gamma$. Similar to Eq. (11), M_v is a function of η and K. We define

$$M_v = \alpha_{R\gamma}(\eta, K). \tag{19}$$

Then

$$M_a = w(2 \times \alpha_{R\gamma}(\eta, K) + L). \tag{20}$$

Eq. (20) gives the space complexity of BFA.

Now, let us consider the time complexity of BFA. Denote by T_a the random variable representing the number of hash function calculations for BFA. Recall that, as shown in Fig. 7, BFA defines three functions, *ProcInbound*, *ProcOutbound*, and *Sample*. Obviously,

$$T_a|\{\textit{Sample} \text{ is executed}\} \equiv 0. \tag{21}$$

When executing Function *ProcInbound*, all the K hash functions need to be calculated. The reason is the following.

1) If variables a and b are both false, Function *BloomFilterInsert* is executed, which calculates K hash functions (see Eq. (12)).
2) Otherwise, at least one of a and b is true; then at least one of the search operations, i.e., *BloomFilterSearch*($RV_{j'}$,$f(p)$), $j' = (I-w+1)\%w, (I-w+2)\%w, \ldots, I\%w$, and *BloomFilterSearch*(IV_I,$f(p)$), returns true. This also means that K hash functions have been calculated (see Eq. (13)).

Therefore, in any case, *ProcInbound* calculates all the K hash functions. Further note that, although *BloomFilterSearch* executes up to $w+1$ search operations, and at most one insertion operation, the total number of hash function calculations in these operations is the same as that in one search operation. This is because the results of hash function calculation in one search operation can be used again by all the other search operations and insertion operation. Therefore,

$$T_a|\{\textit{ProcInbound} \text{ is executed}\} \equiv K. \tag{22}$$

Similarly,

$$T_a|\{\textit{ProcOutbound} \text{ is executed}\} \equiv K. \tag{23}$$

In each time slot, we execute *Sample* once, *ProcInbound* for $R_{pi}\gamma$ times, and *ProcOutbound* for $R_{po}\gamma$ times, where R_{pi} and R_{po} are inbound packet arrival rate and outbound packet

TABLE III
SPACE/TIME COMPLEXITY FOR HASH TABLE, BLOOM FILTER AND BFA

Algorithm	Space complexity	Time complexity
Hash table	M_h (free variable)	Eq. (6)
Bloom filter	Eqs. (10) and (11)	Eqs. (15), (16), and (17)
BFA	Eq. (18), (19), and (20)	Eq. (24)

arrival rate, respectively. Combining Eqs. (21), (22), and (23) and assuming $(R_{pi}+R_{po})\gamma \gg 1$, which is always true in our design of BFA, we have

$$E[T_a] = 0 \times \frac{1}{(R_{pi}+R_{po})\gamma+1} + \frac{K(R_{pi}+R_{po})\gamma}{(R_{pi}+R_{po})\gamma+1} \approx K. \quad (24)$$

Combining Eqs. (24) and (20), we obtain the relationship between M_a and T_a as below

$$M_a = w\left[2\alpha_{R\gamma}(\eta, E[T_a]) + L\right]. \quad (25)$$

Table III lists the space complexity and time complexity for hash table, Bloom filter, and BFA.

4) Numerical Results: In this section, we use the formulae derived in Sections V-A.1 and V-A.3 to compare the hash table scheme with BFA through numerical calculations. The setting of our numerical study is the following:

1) Traces captured from an ISP's edge router shows that the average number of flows during one second is around $250,000$. So, we let R=$250,000$. To reduce the probability of false alarms caused by normal packets with long RTT, we choose Γ large enough such that more than 99% packets have RTT less than Γ. For the same traces, Γ=80 seconds.
2) Suppose we want to detect TCP traffic anomaly. Thus the signature captured from each packet is composed of 32-bit SA, 32-bit DA, 16-bit SP, and 16-bit DP. So $b = 96$ bits.
3) In the BFA algorithm, we use 40 time slots (i.e., $w = 40$), each of which is 2 seconds (i.e., $\gamma = 2$). Also suppose each counter is a 32-bit integer (i.e., $L = 32$).

Fig. 8 shows M vs. $E[T]$ for the hash table scheme, BFA with collision probability 1%, and BFA with collision probability 0.1%. In Fig. 8, X axis represents the time complexity (i.e., the expected number of hash function calculations) and Y axis represents the space complexity (i.e., the number of bits needed for storage). From Fig. 8, we can see that the curve of BFA is below the curve of the hash table. It means BFA uses less space for a given time complexity. Therefore, BFA achieves better space/time trade-off than the hash table. We also see that the curve of BFA with $\eta = 1\%$ is below the curve of BFA with $\eta = 0.1\%$. This shows the relationship between space/time and collision probability. Specifically, to reach a lower collision probability or more accurate detection, we need to either calculate more hash functions or use more storage space.

To see the gain of using BFA, let us look at an example. Suppose $E[T] = 5$, i.e., in each slot, 5 hash function calculations is needed on average. Then, the memory required by the hash table scheme, BFA with $\eta = 0.1\%$, and BFA with $\eta = 1\%$ is $1.01G$ bits, $115.3M$ bits, and $62.9M$ bits, respectively. It can be seen that our BFA with $\eta = 1\%$ can save storage by a factor of 16, compared to the hash table scheme.

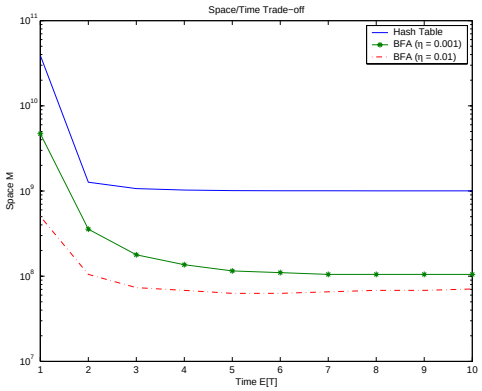


Fig. 8. Space/time trade-off for the hash table, BFA with $\eta = 0.1\%$, and BFA with $\eta = 1\%$

Fig. 8 shows that for the hash table scheme, M_h is a monotonic decreasing function of $E[T_h]$. The observation matches our intuition that the larger table, the smaller collision probability for hash functions, resulting in less hash function calculations. Further note that M_h approaches $R\Gamma(b+1)$ when $E[T_h]$ increases. This is the minimum space required to tolerate up to $R\Gamma$ flows.

For BFA, M_a is not a monotonic function of $E[T_a]$, which approximately equals K. We have the following observations.

- **Case A:** For fixed storage size, the smaller K, the larger the probability that all K hash functions of two different inputs return the same outputs, which is the collision probability. In other words, the smaller K, the larger storage size required to achieve a fixed collision probability. That is, $K \downarrow \Rightarrow M_a \uparrow$.
- **Case B:** Since an input to BFA may set K bits to "1" in a vector V, hence the larger K, the more bits in V will be set to "1" (nonempty), which translates into a larger collision probability. In other words, the larger K, the larger storage size required to achieve a fixed collision probability. That is, $K \uparrow \Rightarrow M_a \uparrow$.

Combining Cases A and B, it can be argued that there exists a value of K or $E[T_a]$ that achieves the minimum value of M_a, given a fixed collision probability. This minimum property can be used to guide the parameter setting for BFA, which will be addressed in Section V-B.1.

B. Optimal Parameter Setting for BFA

This section addresses how to determine parameters of BFA under two criteria, namely, minimum space criterion and competitive optimality criterion.

1) Minimum Space Criterion: According to Eq. (25), three parameters, M_a, $E[T_a]$, and η, are coupled. Since the collision probability η critically affects the detection error rate in our network anomaly detection, a network operator may want to choose an upper bound $\bar{\eta}$ on the acceptable collision

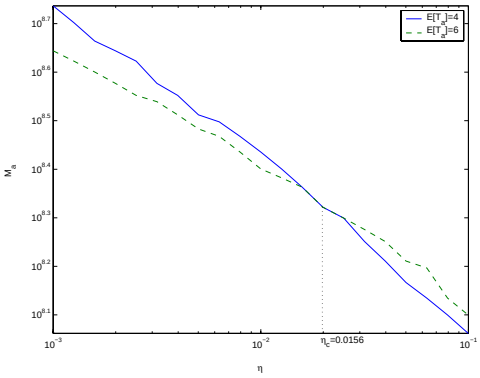


Fig. 10. M_a vs. η for fixed $E[T_a]$.

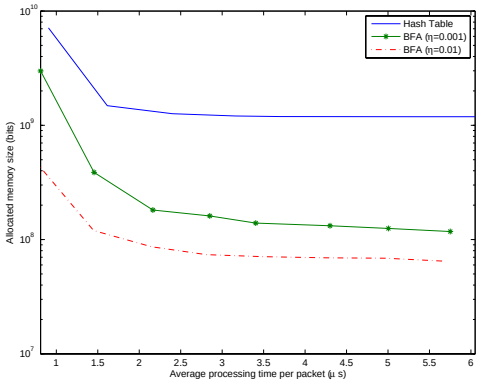


Fig. 11. Memory size (in bits) vs. average processing time per query (in μs)

probability η and then minimize the storage required, i.e.,

$$\min_{E[T_a]} M_a, \quad \text{subject to } \eta \leq \bar{\eta} \tag{26}$$

According to Eq. (25), the solution of (26) is as below

$$M_a^* = \min_{E[T_a]} M_a = \min_{E[T_a]} w\left[2\alpha_{R\gamma}(\bar{\eta}, E[T_a]) + L\right], \tag{27}$$

$$E[T_a]^* = \arg\min_{E[T_a]} M_a = \arg\min_{E[T_a]} \alpha_{R\gamma}(\bar{\eta}, E[T_a]). \tag{28}$$

Fig. 9 shows M_a^* vs. $\bar{\eta}$, and $E[T_a]^*$ vs. $\bar{\eta}$ under the same setting as that in Section V-A.4. From Fig. 9(a), it can be observed that M_a^* decreases with $\bar{\eta}$. This is because the larger collision probability we can tolerate, the less space required. From Fig. 9(b), it can be observed that generally, $E[T_a^*]$ decreases with η. This may be because the smaller $E[T_a^*]$ or K, the larger the probability that all K hash functions of two different inputs return the same outputs, which is the collision probability.

2) Competitive Optimality Criterion: From Eq. (18), it can be observed that η decreases with the increase of M_v if K is fixed; in other words, M_v decreases with the increase of η if K is fixed. Further, from Eqs. (19) and (25), it can be inferred that M_a decreases with the increase of η if $E[T_a]$ is fixed (note that $E[T_a] \approx K$). This is shown in Fig. 10. From the figure, it can be observed that the two lines intersect at a value of collision probability, denoted by η_c. This value is critical for the parameter setting of BFA. If a network operator has a desirable collision probability η, which is greater than η_c, then it should choose $E[T_a] = 4$ since this parameter setting gives both smaller time complexity and smaller space complexity. We call this property 'competitive optimality' since there is no tradeoff between time complexity and space complexity in this case. On the other hand, if a network operator has a desirable collision probability η, which is smaller than η_c, then it needs to make a tradeoff between space complexity and time complexity.

VI. Simulation Results

In this section, we conduct two sets of experiments to show the performance of BFA for feature extraction in high-speed networks. Section VI-A compares the performance of the BFA algorithm with that of the hash table algorithm. In Section VI-B, we show the performance of the complete feature extraction system, which uses the BFA algorithm.

A. BFA Algorithm vs. the Hash Table Algorithm

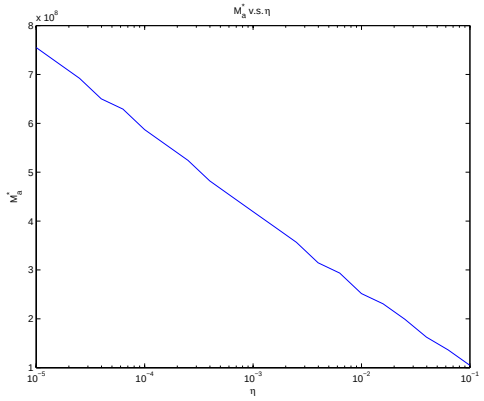
1) Simulation Settings: We apply the hash table algorithm and the BFA algorithm to the time series of signatures extracted from real traffic traces, which were collected by Auckland University [20]. To make a fair comparison with respective to the numerical results in Section V-A.4, we use the same 96-bit signature, i.e., SA, DA, SP, and DP, and let R=250,000 packets/second and Γ=80 seconds, which translates to $250,000 \times 80 = 20M$ input signatures for each simulation. These signatures are preloaded into memory before the beginning of simulations so that I/O speed of hard drive does not affect the execution time of simulations.

For each simulation run of the hash table algorithm, we specify the memory size M_h, and measure the algorithm performance in terms of the average number of hash function calculations per signature query request, denoted by $\hat{T}_h$, and the execution time. Due to the Law of Large Numbers, $\hat{T}_h$ approaches the expected number of hash function calculations per signature query request, i.e., $E[T_h]$ in Eq. (6), if we run the simulation many times with the same M_h. In our simulations, we run the hash table algorithm ten times; each time with a different set of input signatures but with the same M_h.

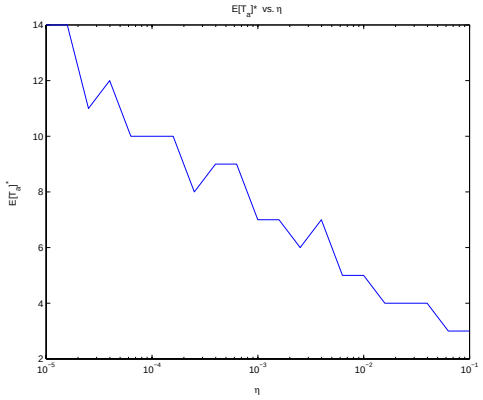
For each simulation run of the BFA algorithm, we specify the memory size m_a and the number of hash functions K, and measure the algorithm performance in terms of the collision frequency, denoted by $\hat{\eta}$, and the execution time. The collision frequency is defined as the ratio of the number of collision occurrences in BloomFilterSearch to the total number of BloomFilterSearch executions. Due to the Law of Large Numbers, $\hat{\eta}$ is a good estimate of collision probability, η.

2) Performance Comparison Between Hash Table and BFA: Fig. 11 shows average processing time per query vs. memory size for the hash table algorithm, BFA algorithm with $\hat{\eta}$=0.1%, and BFA algorithm with $\hat{\eta}$=1%.

From Fig. 11, we observe that 1) compared to the hash table algorithm, the BFA algorithm requires less memory space for the same time complexity (average processing time per query), which was predicted in Section V, and 2) the BFA algorithm with $\hat{\eta}$=1% has a better space-complexity/time-complexity tradeoff than the BFA algorithm with $\hat{\eta}$=0.1% but at cost of higher collision probability, which is predicted by the numerical results in Fig. 8.



(a)



(b)

Fig. 9. (a) M_a^* vs. $\bar{\eta}$, and (b) $E[T_a]^*$ vs. $\bar{\eta}$

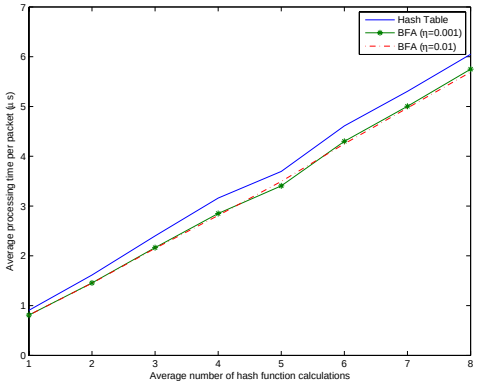


Fig. 12. Average processing time per query (in μs) vs. average number of hash function calculations per query.

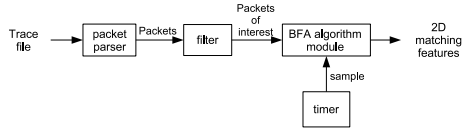


Fig. 14. Feature extraction system.

Fig. 12 shows average processing time per query vs. average number of hash function calculations per query. It can be observed that the average processing time per query linearly increases with the increase of the average number of hash function calculations per query. That is, the larger the average number of hash function calculations per query, the larger the average processing time per query. For this reason, instead of running simulations to obtain the time complexity (i.e., the average processing time per query), in Section V-A, we used the average number of hash function calculations per query to represent the time complexity of the hash table algorithm and the BFA algorithm.

3) Performance Comparison Between Numerical and Simulation Results: Fig. 13 compares the simulations results and the numerical results obtained from the analysis in Section V, for both hash table algorithm and BFA algorithm in terms of space complexity vs. time complexity.

In Fig. 13(a), the numerical result agrees well with the simulation result, except when the average number of hash function calculations per query is close to 1. From Eq. (6), if the expected number of hash function calculations approaches 1, the required memory size approaches infinity; in contrast, simulations with a large M_h may not give accurate results, due to limited memory size of a computer. This causes the big discrepancy between the numerical result and the simulation result when the average number of hash function calculations per query is close to 1. When the average number of hash function calculations per query is greater than or equal to two, it is observed that simulation always requires more memory than the numerical result. This is due to the fact that practical hash function is not perfect. That is, entries in the hash table are not equally likely to be accessed. Hence, Eq. (2) does not hold perfectly, neither does Eq. (3). As a result, the average number of hash function calculations per query in simulation is larger than that predicted by Eq. (6).

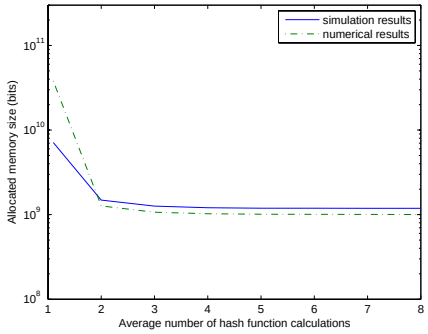
Fig. 13(b) shows that the numerical result agrees well with the simulation result for all the values of the average number of hash function calculations per query under our study.

B. Experiment for Feature Extraction System

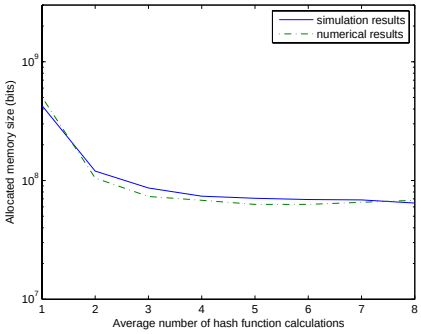
In this section, we show the performance of the complete feature extraction system, which uses the BFA algorithm.

1) Experiment Settings: The feature extraction system used in our experiment is shown in Fig. 14, where the packet parser retrieves a packet from the network trace file stored in a hard disk, the filter selects packets of interest (e.g., selects ICMP packets only for PING flood attack detection), and the timer controls when to produce the features. The reason of conducting this experiment is that we would like to know the performance of the whole feature extraction system that consists of four modules, i.e., the packet parser, the filter, the timer, and the BFA algorithm; in contrast, the experiment in Section VI-A does not involve the interaction among the four modules.

We use the trace data provided by Auckland University [20] as the background traffic. This data set consists of packet header information of traffic between the Internet and



(a) Hash table algorithm



(b) BFA algorithm with η=1%

Fig. 13. Comparison of numerical and simulation results.

Auckland University. The connection is OC-3 (155 Mb/s) for both directions.

In our experiment, we use one 24-hour trace as the background traffic. We simulate network anomalies caused by general flood attacks [13] by randomly inserting TCP packets with random source IP addresses into the the background trace during specified time periods, which are 11000 – 12800 second, 21000 – 24600 second, 62500 – 63700 second, and 70000 – 72400 second. The average attack rate is 1% of the packet rate of the background traffic during the same period.

To detect such general flood attacks, we choose $< SA, DA, SP, DP >$ as the signature of inbound packets and $< DA, SA, DP, SP >$ for outbound ones. Here the 2D feature $|D(\tau_j)|$ is the number of unmatched TCP packets in the jth time slot. The average flow rate is 2480 flows/second. Therefore, we set $R = 2480$. We further set $\eta = 0.1\%$, $K = 8$, $w = 8$, and $\gamma = 10$. Then, by solving Eq. (18) for M_v and requiring M_v to be a power of 2, we obtain $M_v = 2^{15}$ bits. The computer used for our experiments has one 2.4G Hz CPU and 1GB memory.

2) Performance: There are $78,501,441$ packets in the trace. The time to process the data is 296 seconds, including the time to read data from hard drive. The average processing rate is $265,000$ packets/second. Hence, the algorithm can deal with a line rate of 1 Gbps since the average Internet packet size is about 500 bytes. Note that our test is offline and data is read from hard disk, whose access speed is much lower than that of memory. In a real implementation, data is captured by a high-speed network interface and maintained in the memory; so the processing speed can be increased. Furthermore, in our test, a hash function is implemented by software, which is also much slower than a dedicated hardware. Therefore, it is reasonable to anticipate a higher processing rate if a dedicated hardware is used.

Fig. 15 shows the feature extracted from the attack trace. We can see that the 2D matching feature is a good feature that helps distinguish normal conditions from abnormal conditions (with a large number of unmatched packets). More importantly, our BFA can extract the 2D matching feature from a link with a line rate in the order of Gbps.

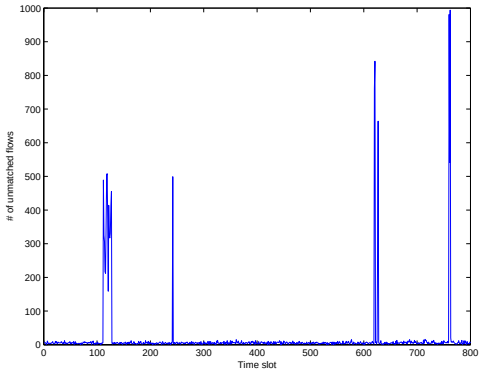


Fig. 15. 2D matching feature: number of unmatched SYN packets

VII. Conclusion

This paper is concerned about design of data structure and algorithms for network anomaly detection, more specifically, feature extraction for network anomaly detection. Our objective is to design efficient data structure and algorithms for feature extraction, which can cope with a link with a line rate in the order of Gbps. We proposed a novel data structure, namely the Bloom filter array, to extract the so-called 2D matching features, which are shown to be effective indicators of network anomalies. Our key technique is to use a Bloom filter array to trade off a small amount of accuracy in feature extraction, for much less space and time complexity. Different from the existing work, our data structure has the following properties: 1) *dynamic* Bloom filter, 2) combination of a *sliding window* with the Bloom filter, and 3) using an insertion-removal pair to enhance the Bloom filter with a *removal* operation. Our analysis and simulation demonstrate that the proposed data structure has a better space/time trade-off than conventional algorithms.

References

[1] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM Press, 2005, pp. 181–192.

[2] "Snort - the open source network intrusion detection system." [Online]. Available: http://www.snort.org/

[3] F. Baboescu and G. Varghese, "Scalable packet classification," *IEEE/ACM Trans. Netw.*, vol. 13, no. 1, pp. 2–14, 2005.

[4] A. Feldmann and S. Muthukrishnan, "Tradeoffs for packet classification," in *IEEE INFOCOM 2000*, vol. 3, no. 26–30, mar 2000, pp. 1193–1202.

[5] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM Press, 1998, pp. 203–214.

[6] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE J. Select. Areas Commun.*, vol. 21, pp. 560–571, May 2003.

[7] D. V. Schuehler, J. Moscola, and J. Lockwood, "Architecture for a hardware based, tcp/ip content scanning system," *IEEE Micro*, vol. 24, pp. 62–69, Jan. 2004.

[8] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM Press, 1999, pp. 135–146.

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

[10] T. Peng, C. Leckie, and K. Ramamohanarao, "Detecting distributed denial of service attacks using source IP address monitoring," Department of Computer Science and Software Engineering, The University of Melbourne, Tech. Rep., 2002. [Online]. Available: http://www.cs.mu.oz.au/~tpeng

[11] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *Proc. ACM SIGCOMM Conference on Internet Measurement '04*, Oct. 2004.

[12] K. Lu, D. Wu, J. Fan, S. Todorovic, and A. Nucci, "Robust and efficient detection of ddos attacks for large-scale internet," *Computer Networks*, vol. 51, no. 18, pp. 5036–5056, Dec. 2007.

[13] J. Mirkovic and P. Reiher, "A taxonomy of ddos attacks and ddos defense mechanisms," in *Proc. ACM SIGCOMM Computer Communications Review '04*, vol. 34, Apr. 2004, pp. 39–53.

[14] J. B. Postel and J. Reynolds, "File transfer protocol," RFC 959, Oct. 1985. [Online]. Available: http://www.faqs.org/rfcs/rfc959.html

[15] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol." *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, June 2000.

[16] F. Chang, W. chang Feng, and K. Li, "Approximate caches for packet classification," in *IEEE INFOCOM 2004*, vol. 4, March 2004, pp. 2196–2207.

[17] R. Rivest, "The md5 message-digest algorithm," RFC 1321, Apr. 1992. [Online]. Available: http://www.faqs.org/rfcs/rfc1321.html

[18] *MD5 CRYPTO CORE FAMILY*, HDL Design House, 2002. [Online]. Available: http://www.hdl-dh.com/pdf/hcr_7910.pdf

[19] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, CA: Morgan Kaufmann, 1998, ch. 5,6.

[20] "Auckland-IV trace data," 2001. [Online]. Available: http://wand.cs.waikato.ac.nz/wand/wits/auck/4/