# A Memory Efficient Anti-Spoofing Method

Hikmat Farhat

Notre Dame University, Computer Science Department,
Zouk Mosbeh, Lebanon
*hikmat.farhat@acm.org*

***Abstract*:** Large-scale denial of service (DoS) attacks present a grave threat to hosts on the Internet. The use of source IP address spoofing makes the situation much worse. The Implicit Token Scheme (ITS), presented in [9], was demonstrated to be an efficient method to defend against IP spoofing. ITS uses the path taken by a packet, which cannot be controlled by the attacker, and binds it to the source IP address of the same packet to form a *token*. All legitimate tokens are stored in a tokens database on border routers. When a packet is received, the border router checks the validity of the token it carries by consulting the tokens database. Only packets carrying valid tokens will be forwarded while the others are dropped. Although very effective, ITS requires border routers to maintain state information for thousands of simultaneous connections which could require more memory than is available on typical routers.

In this paper we add a component to ITS to improve its scalability using Bloom filters. We show that implementing ITS using Bloom filters is simple, saves a substantial amount of router memory, and does not impose large strain on routers. We also modify the basic method to allow for it to be incrementally deployed. The efficiency of the method is demonstrated through simulations by using real-world Internet data.

***Keywords*:** about six key words separated by commas**.**

## 1. Introduction

Today's society has become heavily dependent on the Internet and the services it offers. Mission critical applications have been deployed using the Internet including banking and government transactions. Unfortunately, with the rapid increase in the use of the Internet as a mission critical service came the proliferation of attacks on the Internet infrastructure. The Distributed Denial of Service (DDoS) attacks are one of the most threatening of those attacks. One particular type of DDoS, the flooding attack, floods the link of the victim network with a large amount of packets leading to a high rate of packet drops for legitimate users. Recent studies have shown that the number of DDoS attacks is actually more prevalent than previously thought [14]. What makes protection against IP spoofing paramount is that many approaches that could mitigate DDoS are inefficient in the presence of IP spoofing.

Victims of DDoS attacks cannot authenticate the source address of the received packets because of the destination-based forwarding paradigm of the Internet Protocol (IP). The straightforward method of installing filters at border routers, is rendered inefficient by IP spoofing. It very easy for attackers to choose randomly an IP address as the source for different packets and thus make the protection method infeasible.

Devising methods to detect and block spoofed packets has been actively pursued in the research community. Many solutions have been proposed to the IP spoofing problem are either a variation of the IP traceback method [19,22,28] or try to restrict the address space available to attacker(s) as in [7,16]. A third class of solutions which gained popularity recently is based on the concept of capabilities [1,17].

There is a class of DDoS attacks in which the attackers use intermediate compromised hosts, called zombies or botnets, to mount their attacks [6]. In this class the attackers take control of unwitting hosts and use them later on to mount coordinated DDoS attacks against the victims. In the presence of such class of attacks one could argue that the attackers do not need to spoof the source address because they are not using directly their hosts to mount the attacks and therefore will not reveal their IP addresses. Nevertheless, IP spoofing is still popular as shown in [15]. What is more, we believe that IP spoofing will be used often in the future for many reasons. First, it is more difficult to block spoofed addresses because they don't have a pattern unlike real addresses.

Second, some attacks, like reflector attacks [18] where the attacker poses as some victim to send packets to a number of hosts with the results that the victim receives a large number of replies, rely on IP spoofing to work.

Based on the above discussion about the relative ease in which attackers can fake their source IP address, it remains true that they cannot control the paths taken by packets they send to the victim. This property of being unable to forge the paths taken by packets to reach the target remain one of the cornerstones of defenses against DDoS attacks which employ spoofed packets. In that spirit we have recently proposed the Implicit Token Scheme (ITS) as a method to mitigate DDoS attacks [9].

The key idea in ITS is that attackers cannot complete the TCP three-way handshake if they use spoofed source addresses. The method was demonstrated to be a highly effective defense against spoofed traffic and in subsequent work was shown to be easily deployable on the current Internet infrastructure [10]. The main shortcoming of ITS, however, was its need to maintain state information for many thousands of flows which requires a large amount of router memory. Reducing the memory requirements of ITS is the main contribution of this paper.

Wire-speed filters on Internet routers are usually stored in Ternary Content Addressable Memory (TCAM) which is expensive. Advanced router line cards usually have only 1 TCAM chip which can hold 256k entries to be shared with the router's forwarding table. This limits any DDoS filtering solution to less than 100k simultaneous flows which, for

Received August 20, 2008.

busy sites, is far short of what is expected. Therefore it is paramount for any proposed solution to IP spoofing to be scalable and to save router memory. For more information on router's memory requirements see [3] and references therein.

The rest of the paper is organized as follows. The ITS method is discussed in Section 2. The basic theory of Bloom filters and its adaptation to the ITS method is presented in Section 3. The preliminary implementation and the results of the performed simulations are also presented in Section 4. In Section 4 we discuss the incremental deployment strategies of ITS and the partial matching needed to implement it. We also argue about the benefits of a particular Bloom filter implementation over other strategies. Other work related to DDoS and the application of Bloom filters in networking, are presented in 5. We conclude with Section 6.

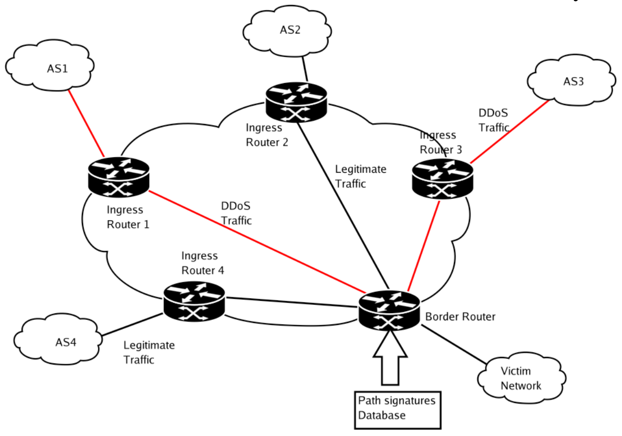

**Figure 1.** Defense model.

## 2. The ITS Method

The Implicit Token Scheme (ITS) provides protection against IP spoofed traffic by having Internet Service Providers (ISPs) install filters at the border router as shown in Figure 1. The installed filters contain *tokens* that should be matched by arriving packets to be forwarded.

The *token* is composed of an IP address and a *path signature*. The *path signature* is a collection of values marked by intermediate routers on the path traveled by the packet from source to destination. These signatures, unlike the source IP address, are uncontrollable by the attacker and therefore cannot be forged. Once the filter(s) is installed a Border router will forward packets if they carry tokens that match entries in the tokens database; otherwise the packet will be dropped. This way IP addresses are tightly bound to unforgeable *path signature*. Furthermore, since the token is 48 bit long the probability of a spoofed address having the correct signature is extremely small.

While the basic technique is straightforward the implementation is not a simple one. The first question that arises is how to collect valid tokens. An obvious solution is to collect tokens during regular traffic when there is no DDoS attack. There are many problems with this approach. First, during a DDoS attack the victim "sees" a large number of previously unseen addresses and all of them are considered spoofed because they are not in the tokens database. Second, due to frequent routing changes the *path signature* most likely will change from the time it is recoded to the time it is used which leads to may false positives. A better approach to building the database is to add the tokens for each TCP session separately after the TCP handshake is completed. This guarantees the integrity of the token because an attacker using spoofed source address cannot complete the TCP handshake.

An added bonus is that the *path signature* is up to date and rarely changes on the order of a TCP transaction. One could reserve a portion of the bandwidth, say, 95% to already established connections which are guaranteed to be non-spoofed, and the remaining 5% to the rest of the traffic to allow for new connections. This method has a serious shortcoming: an attacker can flood the 5% of bandwidth reserved for connection establishment and thus prevents new clients from connecting to the target.

To solve the denial of connection attack we use the concept of a SYN cookie. Originally the SYN cookie was used to protect against SYN attacks. In this work we use it on the Border Router instead of the target. When the Border router receives a TCP SYN segment having a destination address equal to that of the target it responds with a SYN-ACK segment on the behalf of the target. This is done without maintaining state information by using a special value for the Initial Segment Number in the TCP header. For more details on SYN cookies see [25]. When a Border router receives a TCP segment it takes one of the following steps:

- If the segment has SYN=1, it replies with a SYN-ACK that includes the cookie as ISN. This is shown on lines 2-6 in Figure 2.
- If the segment has SYN=0, it checks if the segment contains a valid token then it is forwarded as it shown in Figure 2 on lines 7-9. Otherwise it performs the step below.
- It checks the sequence number. If it is a response to a valid cookie then the token is added to the tokens database and the segment is forwarded. This is shown in Figure 2 on lines 10-13.
- If all of the above checks fail the segment is dropped.

In short, when a border router receives a packet it runs the algorithm shown below.

```
1  for each packet pkt
2      do
3      if pkt.SYN=1
4          then
5              sendCookie
6              Exit
7      if pkt.TOKEN in D
8          then
9              forward packet
10     elseif checkCookie(pkt)=TRUE
11         then
```

```
12            forward pkt
13            insert pkt.TOKEN in D
14       else
15            drop pkt
```

**Figure 2**. Packet Filtering in ITS

The efficiency of the method has been demonstrated in [9] with simulations using real world network data from the Skitter initiative [20]. However, this method requires that Border Routers maintain the tokens database in TCAM memory since filtering operations have to be done at wire speed. The main contribution of this paper is to make the method scalable by reducing its memory requirements. Before doing so we give a few details about *path signatures* that will be needed in later sections.

In the original method the *path signature* was made up of intermediate routers marks where each router contributes 2 bits to the *path signature*. Each 2-bit mark is the result of an MD5 hash of the IP addresses on the current link. When a router with IP address *x* receives a packet from a neighbor with IP address *y* the resulting mark is:

$$mark = MD5\,(x \parallel y)\ \&\ 3 \tag{1}$$

Where || is the concatenation operation and the bitwise AND operation, & 3, is used to retain only the last two bits of the resulting hash. Equation (1) gives the mark contributed by an individual router. It is added to the identification field in the IP header as follows :

$$id_{\text{new}} = id_{\text{old}} << 2 + mark \tag{2}$$

Because the mark is a 2-bit value it is necessary to left-shit by two bits the identification field value to make room for the new mark. This means that the rightmost two bits in the identification field always carry the last mark and the leftmost two bits carry the "oldest" mark. Since the identification field in the IP header is 16-bit long it can hold a maximum of 8 marks. Other sizes of the mark (e.g. 4 or 8 bits) are possible but it is argued in [9,26,29], based on the average "Internet length" that two-bits are optimal.

## 3. Bloom Filters

As can be seen from lines 7 and 13 in the algorithm presented in Figure 2 , packet tokens need to be stored and retrieved from the tokens database at wire speed. This necessitates the use of the expensive and limited size of on-chip SRAM on the Border Router. To reduce the memory footprint we will use Bloom filters to store the tokens database. What follows is a quick overview of Bloom filters.

Bloom filters were introduced in 1970 by B. H. Bloom [4]. They have been widely used since, especially in database applications. Recently there has been a surge in the use of Bloom filters in networking applications (see [5] for a survey).

A Bloom filter is a space-efficient data structure used to test set membership. It is an array of *m* bits, initialized to zero, used to represent a set of *n* elements, $S=x_1,\dots\ x_n$. The filter uses *k* independent and uniform hash functions, $h_1,\dots,h_k$, each with range in $1,\dots,m$. To "add" an element $x_i$ in $x_1,\dots,x_n$ to the filter the *k* hash functions are applied to $x_i$ and the corresponding bits in the filter are set to one. Adding an element *x* to the filter is written in pseudo-code as follows:

```
ADD-ELEMENT(x)
1 for j=1 to k
2 do
3     filter[$h_j(x)$] gets x
```

It is clear that when a particular bit is set, an additional setting does not change it. To check if an element *y* belongs to the set the *k* hash functions are applied to *y* and the corresponding bits are checked. If one of the bits is 0 then clearly the element is not in the set. If all the bits are equal to 1 then we could say that the element belongs to the set. The following pseudo-code checks if *y* is an element of the set:

```
CHECK-ELEMENT(y)
1 for j=1 to k
2 do
3     if filter[$h_j(y)$]=1
4         then return FALSE
5 return TRUE
```

Obviously, an element *z* could have all the corresponding bits equal to 1 without the element itself belonging to the set. This is called a *false positive*. It is in our interest that the rate of *false positives* be as small as possible. The *false positive* rate can be calculated as follows. When a given hash function $h_i$ is applied to an input $x_1$ the results is a value between 1 and *m*. Since the hash functions are uniform, the probability that this result is equal to a particular number *v* is 1/*m*. Therefore the probability of the bit at position *v* being 1 after one hash function is 1/*m*. The probability that it is 0 is 1-1/*m*. The probability that it is 0 after all *k* hash functions are applied is $(1-1/m)^k$. Since there are *n* elements in the set, the probability that the bit *v* is equal to 0 after we process all elements is $(1-1/m)^{kn}$. Hence 1- $(1-1/m)^{kn}$ is the probability that a given bit *v* is set to 1 after all input elements $x_1,\dots,x_n$ are processed. Since we want the false positive rate, we need the probability that for an arbitrary input *y* the corresponding *k* bits are 1 without *y* belonging to the set. This probability is

$$f_p = \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k \tag{3}$$

$$\approx \left(1-e^{\frac{nk}{m}}\right)^k \tag{4}$$

Asymptotically the false positive rate depends on $k$ and the ratio $m/n$. If we fix the ratio $m/n$ then one can show [13] that the minimum of the false positive rate in equation (4) as a function of $k$, occurs when

$$k_0 = \frac{m}{n} \ln 2 \tag{5}$$

And the optimal false positive ratio is

$$f_{p_0} = \left(\frac{1}{2}\right)^k \tag{6}$$

Usually, the false positive rate and the number of elements $n$ are fixed and we need to deduce the number of bits required to achieve those values. Combining equations (5) and (6) we get

$$m = -2.08 n \ln f \tag{7}$$

One disadvantage of Bloom filters is that it is not possible to delete entries stored in the filter. To do so requires the setting to zero all the $k$ bits that the entry points to. But this could confuse the filter since as we mentioned a bit could be set to 1 by multiple entries. To solve this problem a variation of Bloom Filters called counting Bloom Filters was introduced by Fan et. al. [8]. In a counting Bloom Filter each entry is a counter rather than a single bit. When we add an entry the corresponding counters are incremented and when the item is removed the corresponding counters are decremented.

In fact, only 4 bits per counter are what mostly application need [8]. To simplify the discussion we will ignore this aspect in the rest of the paper and consider only connection establishment, not connection closure. This is not really a restriction since we could assume that once a DDoS attack is over the Border Router resets all its entries.

## 3.1 Building the filter

Originally the list of tokens was stored in what is called a tokens database. The implementation of the database was not specified but rather assumed to exist. Furthermore, an assumption was made that one can retrieve and store entries in the database. In this section we show how the above-mentioned database can be implemented as a single bloom filter.

Each entry in the database contains a token, which is composed of the source IP address and the corresponding *path signature* stored in the 16-bit IP identification field of the IP header [9]. This field is marked by the routers along the path, from the source to the destination, where each router contributes 2 bits.

In the discussion of Bloom filters in Section 3 we have assumed that the elements of the set and their number are known in advance. In ITS, the tokens are added to the filter every time a TCP connection is established. Therefore the number of elements is not known in advance but increases with time. This is not really a problem at all. Recall from Section 3 that the number of bits needed to get the optimal value of false positive is proportional to $n$, which is the number of elements in the set.

The pseudo-code for adding the token of a packet to the filter is shown in Figure 3 below:

```
ADD-PACKET(pkt)
1 token=pkt.sig|| pkt.source
2 for $i$=1 to $k$
3    do
4       bitPos=$h_i$(token)
5       filter[bitPos] gets 1
```

**Figure 3**. Adding a packet token to the filter

Similarly checking if a packet is in the filter is shown in Figure 4.

```
CHECK-PACKET(pkt)
1 token=pkt.sig|| pkt.source
2 for i=1 to k
4    do
5       bitPos=$h_i$(token)
6       if filter[bitPos]=0 return FALSE
7 return TRUE
```

**Figure 4**. Checking if a packet is stored in the filter

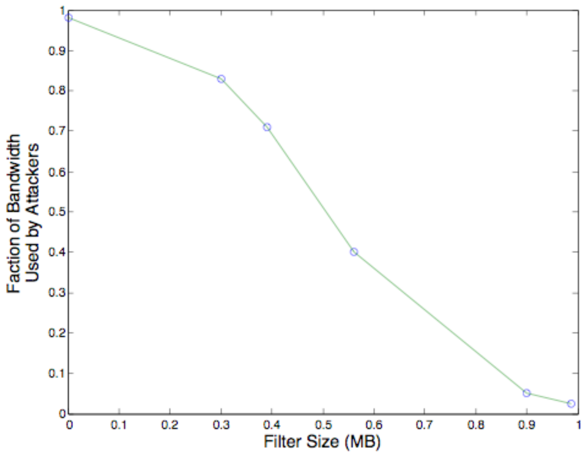

**Figure 5**. The efficiency of the filter as a function of the filter size

In our analysis we will regard this number $n$ as an upper bound on the number of elements that we can store in the Bloom filter. It can be seen from equation (4) that the smaller the value of $n$ the smaller the false positive rate.

It should be noted that it is possible in the algorithm shown in Figure 4 above that a packet will have all the resulting bits equal to 1 without the packet actually being in the filter. Having implemented these two functions using a Bloom filter we can use them in the algorithm shown in Figure 2 to replace the original functions. The function CHECK-PACKET replaces the condition of the if statement on line 7 in Figure 2 and the ADD-PACKET function replaces the insert statement on line 13 in the same Figure.

## 3.2 Implementation

Due to its widely availability and effectiveness we have chosen to use MD5 for hashing. The 128-bit output of an MD5 was used as four independent 32-bit hashes therefore we needed two MD5 operations to generate the eight hash functions $h_1,\ldots,h_8$. Our goal is to maintain about 500,000 flows. For a false positive rate of 1%, from equation (7) the filter size is 2.08 X 5 X $10^5$ ln 0.01≈0.6 MB. As a comparison, without Bloom filters we need 6 bytes for each token for a total of 6 X 5 X $10^5$= 3MB. This is a memory saving of 5 times. We have performed a series of simulations using real-world topological data from Skitter [20]. For every trial run we chose randomly 600 hosts: 100 were used as clients and 500 as attack sources. The IP address and path identifier of clients were manually added to the Bloom filter (not through TCP). The attacking sources send data at the constant rate of 10M packets/s while the clients send at the rate of 1M packets/s. The link victim's link rate is set to 100MB, i.e. just enough for the legitimate clients. The metric used to measure the performance of our method is the fraction of bandwidth of the link between the border router and the target consumed by the attacking packets. As expected, the results in Figure 5 show that the bigger the filter size, the better the efficiency of the method since the *false positive* rate is smaller.

It should be noted that for a size of 0.9 MB less than 5% of the bandwidth is used by the attackers which is an excellent results with a gain of a factor of more than 3 in memory size since the original ITS method requires 3 MB of memory.

## 4 Incremental Deployment

One cannot expect that all routers on the Internet deploy ITS at the same time. Any method would be useless if it cannot be incrementally deployed. The original ITS method was shown to be incrementally deployed [10]. The question here is to do the same but with a smaller memory footprint using Bloom filters. To be able to do that it is helpful to describe how the original method worked. Suppose a given Border Router receives two packets, $p_1$ and $p_2$ with signatures $sig_1=a_{15}\ldots a_0$ and $sig_2=b_{15}\ldots b_0$ . The 16-bit signature is stored in the identification field in the IP header. The base ITS method uses exact match to compare packets: two $sig_1=sig_2$ are equal if $a_i=b_i$ for all *i*. Exact match cannot be used if we take incremental deployment into account. When ITS is incrementally deployed, the packets will be forwarded by routers that do not implement ITS. The marks of these routers will be missing from the packet signature.

### 4.1 Partial Matching

It is helpful to illustrate with an example the idea of partial deployment. Let *s* and *d* be the source and target respectively. As required by the Internet Protocol, every packet sent from *s* to *d* has to have a different value in the identification field in the IP header. Assume further that there are five intermediate routers $R_0$, $R_1$, $R_2$, $R_3$ and $R_4$ between *s* and the Border Router that protects *d*. Suppose that one of the intermediate routers, say $R_2$, does not implement ITS. Let $M_0$, $M_1$, $M_2$, $M_3$ and $M_4$ be the marks of the routers respectively. Initially, *s* needs to establish a TCP connection with the target *d*. This is done via the Border Router, which saves the *path signature* in the tokens database. It is important to note that signature saved by the Border Router depends on the original value of the identification field and the path taken by the packet. In our example suppose that the initial value of the identification field when the TCP handshake is completed by *s* (this is when the Border Router saves the signature) is $a_{15}\ldots a_0$. Since each router mark consumes 2 bits and the identification field in the IP header is updated according to equation (2) then when the packet reaches the Border Router it has the value $a_8\ldots a_0M_4M_3M_1M_0$. Note that since $R_2$ does not implement ITS its mark is absent. At a later time when *s* sends a packet with initial value for the identification field equal to $b_{15}\ldots b_0$$ it will reach the border router with the signature $b_8\ldots b_0M_4M_3M_1M_0$.

Clearly the two signatures are not the same and the border router drops the second packet. In fact the Border Router drops all the packets subsequent to connection establishment because all of them will have different initial value as required by IP and therefore will reach the Border Router with different *path signature* from the one stored in the tokens database. The example we have provided is not a rare occurrence. In fact to minimize this problem we have chosen that each router mark should be 2-bits. As can be seen from Figure 6 the number of paths that have length (number of hops) more than 8 is very small.

The solution to the above problem is to use partial matching instead of exact matching [10]. The basic idea in partial matching is to count the number of identical marks from right to left in the *path signature*. In the example above the signatures $sig_1=a_8\ldots a_0M_4M_3M_1M_0$ and $sig_2=b_8\ldots b_0M_4M_3M_1M_0$ have at least 4 identical marks. Starting from right to left the identical marks are: $M_0$ then $M_1$

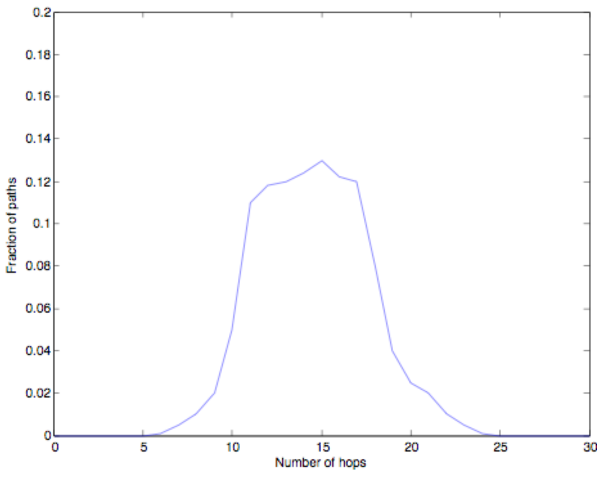

**Figure 6**. The distribution of the paths length.

then $M_3$ and finally $M_4$. The algorithm for partial matching is shown in Figures . The two signatures used in the example could have more matches (accidental) depending on the values of the a'i and b'i. Once the number of matches is computed it is used as a priority, which is then assigned to the packet. Therefore in this method no packet is dropped, it is assigned a low priority.

```
COUNT-MATCHES(sig,pkt)
1 count ← 0
2 for i=1 to 8
3   do
4     if pkt.sig & $(2^{2i}-1)$= sig & $(2^{2i}-1)$
5       then
6         count ← count+1
7       else return count
8 return count
```

**Figure 7**. Counting the number of matches between two signatures

```
1 for each packet pkt
2   do
3     if pkt.SYN=1
4       then
5         sendCookie
6         Exit
7     if checkCookie(pkt)=TRUE
8       then
9         add pkt to queue 0
10          insert pkt.token in D
11      else
12          sig=lookup(pkt)
13          n ← COUNT-MATCHES(pkt)
14          add pkt to queue n
```

**Figure 8**. Modified packet filtering using partial matching

### 4.2 Partial Matching Using Bloom Filters

Now we need to implement partial matching using Bloom filters to save memory. The approach is similar to what was done before but quite. The main obstacle is that the hash of the concatenation of two strings is *not* equal to the concatenation of the hashes. In other words, given a hash function $g$ and two strings $x$ and $y$, in general

$$g(x||y) \neq g(x) \,||\, g(y)$$

For a given IP address $IP_x$ and *path signature* $sig_x$, instead of considering the whole token, $IP_x \,||\, sig_x$, for hashing operations we consider intermediate values of the token. If we consider again the example given in Section 41. Assume that the source with IP address $IP_x$ sent a packet with the identification field having the value $a_{15}\ldots a_0$ and the packet is forwarded by 4 ITS routers with marks $M_4$, $M_3$, $M_1$ and $M_0$ (recall that the router with mark $M_2$ does not implement ITS). We know that the *path signature* of the packet when it reaches the Border Router will be $a_8\ldots a_0M_4M_3M_1M_0$. The problem arises because once the token is hashed we cannot perform partial matching. Consider the hash of the whole token with a hash function $g$:

$$g(IP_x \,||\, a_1a_0M_4M_3M_1M_0)$$

Clearly we cannot perform any partial matching on the above value. To be able to perform partial matching it important to perform the hashing on the *partial signatures*. We use $k$ hash functions but the hashing is applied differently. For a given source IP address $IP_x$ and *path signature* $sig_x$, we perform $k$ different hash operations on 8 modifications of the packet token (for a total of 8*$k$ hash operations per packet):

$$h_i(IP_x \,||\, sig_x \,\&\, (2^{2j}-1)) \qquad 1\leq i \leq k,\ 1\leq j \leq 8$$

Where again || is the concatenation operator and & is the bitwise AND operator. For example, the first $k$ hashes corresponding to $j$=1, give $h_i(sig_x \,\&\, 3) = h_i(IP_x|| M_0)$ with $0< i < k+1$ because $sig_x\& 3=M_0$. We say that two *path signatures* $sig_1$ and $sig_2$ have a match of *order j* if and only if for all $i$ we have:

$$h_i(IP_1 \,||\, sig_1 \,\&\, (2^{2j}-1))=h_i(IP_2 \,||\, sig_2 \,\&\, (2^{2j}-1))$$

Note in the above equation we have used the *same* source IP address. We illustrate the idea by applying it to the example given in Section 4.1. Recall that the same source, with IP address $IP_x$, sent two packets with different initial values in the identification field: $a_{15}\ldots a_0$ and $b_{15}\ldots b_0$. The ITS router marks are $M_4$, $M_3$, $M_1$, and $M_0$. Using hash function $h_i$ we get the following set of values:

$$h_i(IP \,||\, M_0) \qquad h_i(IP \,||\, M_0)$$
$$h_i(IP \,||\, M_1M_0) \qquad h_i(IP \,||\, M_1M_0)$$
$$h_i(IP \,||\, M_3M_1M_0) \qquad h_i(IP \,||\, M_3M_1M_0)$$
$$h_i(IP \,||\, M_4M_3M_1M_0) \qquad h_i(IP \,||\, M_4M_3M_1M_0)$$
$$h_i(IP \,||\, a_1a_0M_4M_3M_1M_0) \qquad h_i(IP \,||\, a_1a_0M_4M_3M_1M_0)$$

Where in the above $1\leq i \leq k$. Clearly, the result of the first four lines are identical, even though the two packets had different identification field initially. This means that the two signatures have matches of order 0, 1, 2 and 3.

As before we use the number of matches between signatures to assign a priority to a packet. We assign a priority $n$ depending on the *match order* of a signature. If a signature have match orders 0…$k$ then it is assigned priority $k$+1. In the example above the number of matches would be four, because it has matches of order 0,1,2 and 3. This matching procedure is shown in the algorithm in Figure 9. Furthermore, given a border router, for every packet it receives it executes the algorithm in Figure 10.

```
HASHED-MATCHES(pkt)
1 count ← 0
2 for i=1 to 8
3   do
4     mask=pkt.sig & $(2^{2i}-1)$
5     token=pkt.IP || mask
6     for i=1 to k
7       do
8         bitPos=$h_i$(token)
9         if filter[bitPos]=0
10            then return count
11          count ← count+1
12    return count
```

**Figure 9.** Counting the number of hashed matches of a

packet.

```
1  for each packet pkt
2     do
3        if pkt.SYN=1
4           then
5              sendCookie
6              Exit
7        if checkCookie(pkt.ack)=TRUE
8           then
9              add pkt to queue 0
10                ADD-PACKET-TO-FILTER(pkt)
11          else
12                n ⟵ HASHED-MATCHES(pkt)
13                add pkt to queue n
```

**Figure 10**. Modified packet filtering using hashed partial matching

Similar to the case of exact matching it is clear that it is possible to have a partial matching without the original signatures being the same. These false positives are expected when using Bloom filters.

### 4.3 Implementation Details

It turns out that the straightforward implementation as shown in Figures 9 and 10 is not very efficient. In fact we would need 8 "independent" Bloom filters, one for each signature variation. If we check the results of Figure 5 then we could see that to achieve 70% bandwidth consumption by attackers (only 30% reserved for legitimate users) we need a filter with size equal to 0.4MB which means for the 8 filters we need about 3.2MB which is larger than required *without* using Bloom filters.

A better approach is to use 8 hash functions, one hash function for every *variation*. Not only the algorithm for counting the number of signature matches in Figure 9 needs to be changed but also we need to include a function to add a given token to the filter as this is not straightforward as in the case of using 8 filters. As in all cases, adding a token to the filter is done *after* the TCP handshake is completed and the Border Router checks the validity of the SYN-cookie. Given a source IP address, $IP_x$ and *path signature* $sig_x$, and the 8 hash functions labeled $h1 \ldots h_8$, the Border Router uses the algorithm shown in Figure 11 to add the packet token to the filter and the one shown in Figure 12 to count the number of matches in the filter.

```
ADD-PACKET-TO-FILTER(pkt)
1  for i=1 to 8
2     do
3        mask=pk.sig & ($2^{2i}$-1)
4        token=pkt.IP || mask
5        bitPos=$h_i$(token)
6        filter[bitPos] ⟵ gets 1
```

**Figure 11**. Adding the hash of a packet variation to the filter

```
MODIFIED-HASHED-MATCHES(pkt)
1  count ⟵ 0
2  for $i$=1 to 8
3     do
4        mask=pkt.sig & ($2^{2i}$-1)
5        token=pkt.IP || mask
8        bitPos=$h_i$(token)
9        if filter[bitPos]=0
10             then return count
11          count ⟵ count+1
12       return count
```

**Figure 12.** Modified version of the function in Figure 9.

On the surface it looks like the number of hash operation is going to be as before, 8 per packet. In reality, as we mentioned in Section 3.2 we used only two MD5 operations to implement the 8 hashes. In this case this cannot be done since we are applying the hashing to 8 different variations and therefore we need to do eight hash operations per packet instead of only two. If hashing becomes a bottleneck, instead of MD5 one can use other hashing techniques have been demonstrated to perform well and are much cheaper to implement in hardware [23].

To test the efficiency of our method we performed a series of simulations using the real-world Internet paths provided by Skitter initiative [20].

First we characterize partial deployment by a parameter $d$ which is the number of routers implementing our method for a given path. Given a path containing $k$ routers, we randomly choose $d \le k$ routers to implement the method and the remaining $k$-$d$ routers forward packets in the normal fashion without modifying the packet headers. Figure 13 shows the results of the simulation for $d$=1, $d$=2, and $d$=4. For comparison purposes we included in Figure 13 the full deployment results already shown in Figure 5 It is clear from the presented results that the proposed method succeeded in reaching the goals we have set for it: combating IP spoofing

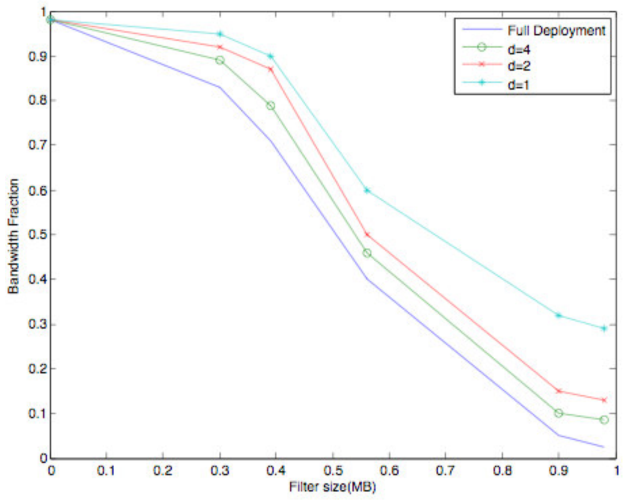

**Figure 13.** Bandwidth fraction used by attackers for three different values of *d* as well as the case of full deployment

while minimizing the memory usage on routers. We can see from Figure 13 that even if only one router along the path (*d*=1) about 70% of the bandwidth is reserved to legitimate users with a cost of 1MB of router memory. The results for *d*=1 is important because it is an excellent incentive for Internet Service Providers (ISP) to deploy ITS. This means that even if no other ISP deploys ITS they still get an efficient method to protect their networks from IP spoofing.

## 5 Related Work

The earliest work to solve the problems created by the ability of attackers to spoof the source IP address of packets are IP traceback techniques [19,22,24] which permit a target to trace the origin of packets even if source address spoofing is employed. One method in particular uses Bloom filters to minimize the storage requirements on routers [21]. Unlike our approach, most these methods use probabilistic packet marking. They require routers to add, with a certain probability, a mark to the IP header. The path taken by attack packets is reconstructed when a sufficient number of attack packet has been received by the victim. These methods have been shown to be successful in finding the approximate origin of the attack packets. The cost of the reconstruction algorithm, however, becomes prohibitive when the number of attackers is large.
Other approaches to source address spoofing and one of the earliest such methods is Ingress filtering by Ferguson and Senie [11]. This requires the installation of ingress filtering at every ISP. Even so, IP addresses in the local network can still be spoofed. Another approach to ingress filtering is the SAVE protocol proposed by Li. et. al. [12].
The information obtained from the Border Gateway Protocol (BGP) update message was used by Duan et. al. [7] to selectively drop packets that appear to be spoofed. Also based on BGP updates is the method proposed by Park and Lee [16] to discard spoofed IP packets using a route-based detection method. The problem with BGP-based methods is the need for independent Autonomous Systems (AS) to cooperate when in fact they have no incentive to do so.
To our knowledge, the first use of deterministic packet marking was introduced by Yaar et. al. in [26] and was extended in [29]. They used the path identification which is a deterministic mark stamped by the intermediate routers on every packet as a way to distinguish malicious from legitimate users. Even if one assumes that the malicious signatures can be clearly identified the number of malicious and legitimate users having the same signature grows as the number of attackers grows which quickly leads to self-inflicted DoS.
Following the introduction by Anderson et. al. [1] of the concept of capabilities there was a flurry of papers published on the subject [17,27,30]. Argyraki et. al. [2] argued that capabilities are not necessary nor sufficient to defend against DDoS attacks . Their main contention is that these methods are prone to denial of capabilities attacks.

## 6 Conclusion

The Implicit Token Scheme (ITS) is an efficient method to defend against spoofed IP traffic. In this paper we have proposed the use of Bloom filters, a space efficient data structure, to store rules of ITS and thereby reduce the storage requirements on intermediate routers. Since Bloom filters can give rise to *false positives* we also derived an expression for the *false positive* rate as a function of the filter size as well as the optimal values needed to minimize the rate of *false positives*. Several simulations were preformed on real-world data and the results prove that the proposed method accomplishes its aim of saving the (up to a factor of 5) memory requirements on intermediate routers. Even in the case of partial deployment the simulation results show that the method is still effective both in combating IP spoofing and saving on router memory.

## References

[1] Anderson, T., Roscoe, T., and Wetherall, D. "Preventing internet denial-of-service with capabilities". *SIGCOMM Comput. Commun. Rev.*, pp. 39-44, 2004.
[2] Argyraki, K., and Cheriton., D. "Network capabilities: The good, the bad and the ugly". In *HotNets-IV: The Fourth Workshop on Hot Topics in Networks*, pp. 27-32, 2005.
[3] Argyraki, K., and Cheriton, D. R. "Active Internet Traffic Filtering: Real-time Response to Denial-of-service Attacks". In *Proceedings of the Annual USENIX Technical Conference,* pp. 135-148, 2005.
[4] Bloom, B. H. "Space/time Trade-offs in Hash Coding With Allowable Errors", Communications of the ACM, pp. 422-426, 1970.
[5] Broder, A., and Mitzenmacher, M. "Network Applications of Bloom Filters: A Survey", Internet Mathematics, pp. 485-509, 2005.
[6] Cooke, E., Jahanian, F., and McPherson, D "The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets". In SRUTI'05: Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Workshop, pp. 39-44, 2005.

[7] Duan, Z., Yuan, X., and Chandrashekar, J. "Constructing Inter-domain Packet Filters to Control IP Spoofing Based on BGP Updates". In Proceedings of IEEE INFOCOMM, pp. 1-12, 2006.

[8] Fan, L., Cao, P., Almeida, J., and Broder, A. Z. "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol", IEEE/ACM Trans. Netw., pp. 281-293, 2000.

[9] Farhat, H. " Protecting TCP Services From Denial of Service Attacks". In *Proceedings of the ACM SIGCOMM workshop on Large-scale attack defense*, pp. 155-160, 2006.

[10] Farhat, H. "An Effective Defense Against Spoofed IP Traffic". In *NTMS'2007: Proceedings of the First IFIP International Conference on New Technologies*, Mobility and Security, pp. 375-384, 2007.

[11] Ferguson, P., and Senie, D. "Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing". RFC 2827, 2000.

[12] Li, J., Mirkovic, J., Wang, M., Reiher, P., and Zhang, L. "SAVE: Source Address Validity Enforcement Protocol". In *Proceedings of IEEE INFOCOMM*, pp. 1557-1566, 2002.

[13] Mitzenmacher, M. "Compressed Bloom Filters", *IEEE/ACM Trans. Netw.,* pp. 604-612, 2002.

[14] Moore, D., Shannon, C., Brown, D. J., Voelker, G.~M., and Savage, S. "Inferring Internet Denial-of-service Activity". *ACM Trans. Comput. Syst*., pp. 115-139, 2006.

[15] Pang, R., Yegneswaran, V., Barford, P., Paxson, V., and Peterson, L. "Characteristics of Internet Background Radiation". *In IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pp. 27-40, 2004.

[16] Park, K., and Lee, H. "On the Effectiveness of Route-based Packet Filtering for Distributed DoS Attack Prevention in Power-law Internets". In *Proceedings of ACM SIGCOMM*, pp. 15-26, 2001.

[17] Parno, B., Wendlandt, D., Shi, E., Perrig, A., Maggs, B., and Hu, Y.-C. "Portcullis: Protecting Connection Setup From Denial-of-capability Attacks". In *Proceedings of ACM SIGCOMM*, pp. 289-300, 2007.

[18] Paxson, V. "An Analysis of Using Reflectors for Distributed Denial-of-service Attacks". *Comput. Commun. Rev.*, pp. 38-47, 2001.

[19] Savage, S., Wetherall, D., Karlin, A., and Anderson, T.} "Network Support for IP Traceback". *IEEE/ACM Trans. Netw.*, pp. 226-237, 2001.

[20] CAIDA's skitter initiative. http://www.caida.org.

[21] Snoeren, A., Partridge, C., Sanchez, L. A., Jones, C. E., Tchakountio, F., Schwartz, B., Kent, S. T., and Strayer, W. T. "Single-packet IP Traceback". *IEEE/ACM Trans. Netw.,* 721-734, 2002.

[22] Song, D., and Perrig, A. "Advanced and Authenticated Marking Schemes for IP Traceback". In *Proceedings of IEEE INFOCOMM*, pp. 878-886, 2001.

[23] Stone, J., Greenwald, M., Partridge, C., and Hughes, J. "Performance of Checksums and CRC's Over Real Data". *IEEE/ACM Trans. Netw*., pp. 529-543, 1998.

[24] Sung, M., and Xu, J. "IP Traceback-based Intelligent Packet Filtering: A Novel Technique for Defending Against Internet DDoS Attacks". In *Proceedings of the IEEE International Conference on Network Protocols*, pp. 302-311, 2002.

[25] D.J. Bernstein. http://cr.yp.com/syncookies.html.

[26] Yaar, A., Perrig, A., and Song, D. "PI: A Path Identification Mechanism to Defend Against DDoS Attacks". In *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 93-107, 2003.

[27] Yaar, A., Perrig, A., and Song, D. "SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks". In Proceedings of the IEEE Symposium on Security and Privacy, pp. 130-143, 2004.

[28] Yaar, A., Perrig, A., and Song, D. "FIT: Fast Internet Traceback". In *Proceedings of IEEE INFOCOMM*, pp. 1395-1406, 2005.

[29] Yaar, A., Perrig, A., and Song, D. "StackPi: New Packet Marking and Filtering Mechanisms for DDoS and IP Spoofing Defense". *IEEE Journal on Selected Areas in Communications*, pp. 1853-1863, 2006.

[30] Yang, X., Wetherall, D., and Anderson, T. "A DoS-limiting Network Architecture", Comput. Commun. Rev., pp. 241-252, 2005.

**Author Biography**

Hikmat Farhat is an assistant professor in the Computer Science Department , Notre Dame University, Lebanon. His research interests are Computer Networks and Network security.