Network Access Control using Bloom filters

Parvez Anandam

March 12, 2007

1 Introduction

As the corporate network becomes increasing porous due to the proliferation of mobile devices such as laptops and PDAs, there is a growing need to protect vital enterprise servers against internal and external attacks. The current regulatory climate, particularly in heathcare and corporate accounting, is another strong incentive for allowing only authorized personel to access key applications.

These trends, combined with rapidly increasing network line speeds, place great demands on a centralized point of access control. A single point of control is very appealing to system administrators and security officers for many reasons. However, naïve implementations of policy engines simply fail to keep up with the amount of network traffic being inspected and are a serious bottleneck.

In this paper, we explore the use of Bloom filters [1] to achieve better spacial and temporal scaling. Bloom filters have been used with spectacular success in many areas of networking [2]. In particular, they have proved very useful when used in a closely related application: deep packet inspection for the purpose of stopping threats such as viruses [3].

2 Problem definition

Consider a central policy enforcement point that protects many web applications and web documents (henceforth called *resources*). Each web application or document is referenced in the standard way, by the use of a URL (e.g. http://hr.acme.com/forms/hire.html and http://eng.acme.com/auto/design.pdf). There are many users of the system, and each user or group of users is allowed access to a particular subset of the resources.

There are two approaches to access control:

"Default Permit": All users have access to all resources unless they are explicitly denied access. Associated with each user or group of users is a list of URLs to resources they are denied access to. We will call such a list a "Deny ACL" (Deny Access Control List).

"Default Deny": All users have access to no resources unless they are explicitly granted access. Associated with each user or group of users is a list of URLs to resources they are permitted access to. We will call such a list a "Permit ACL".

Both approaches are used in practice. The "Default Permit" policy is often easier to configure whereas the "Default Deny" policy is perceived as more secure.

The number of URLs in ACLs can easily become quite large (thousands of URLs are common, hundreds of thousands occur sometimes). While this amount of space is not a big concern in user space applications, it starts becoming one the closer one gets to the hardware (e.g. in OS kernels, FPGAs) where faster processing times can be achieved.

A non-probabilistic algorithm for access control checks must store all the URLs verbatim. Therefore, such an algorithm is impractical at the scales mentioned. A common approach used in deep packet inspection [3] is to take a two-phased approach. The first phase is to use a probabilistic filter that does a good job of filtering out most of the inspected data as non-matches. It only lets a small fraction (say 1%) through to the second phase, which performs a slow, non-probabilistic check to determine whether there is a match between the data and malicious signatures. The first phase can then be implemented as close to the hardware as desired, while the second phase is implemented in software.

Such an approach can be successfully used in our "Default Permit" case, as we shall see. It is less obvious whether such a strategy can be successfully applied to the "Default Deny" case but we explore the possibility.

3 Canonical set-membership algorithms

Before we look at probabilistic algorithms, it is instructive to look at the standard alternatives.

A canonical solution to the set membership problem is to use a balanced binary search tree. The lookup, insertion and deletion operations all run in $O(\log n)$ time. The space needed to store n elements, each of size m bits, is $O(mn)$.

Hash tables are also commonly used to represent sets. Lookup, insertion and deletion run in $O(1)$ time on average but take $O(n)$ time in the worst-case. Again, the space needed to store n elements, each of size m bits, is $O(mn)$.

The primary drawback of these standard solutions is that the individual elements must be stored as is, in some form. There is no escaping the $O(mn)$ space requirement, unless one is willing to tolerate a certain number of false positives [4].

It is worth mentioning that there is another common probabilistic approach, besides the Bloom filter. It is to use a hash function without storing any of the elements. In this case, the search time is $O(1)$ as with the hash table, but the space needed is $O(n \log n)$. We have now introduced a false positive rate of $1/n$. Hashing is in fact simply a Bloom filter with one hash function. In the asymptotic limit, the false positive rate is made vanishingly small by just using $\Theta(\log n)$ bits per element. Therefore, hashing has great theoretical appeal. In practice however, space considerations often dominate and using a constant number of bits per element (as in the Bloom filter), at the expense of a higher false positive rate, is a good trade-off.

4 Formal definition of a Bloom filter

In 1970, Bloom proposed [1] a very elegant probabilistic data structure for testing set-membership. His original application was automatic hyphenation using an English dictionary. Core memory (RAM) was fairly limited at the time, therefore the entire dictionary of the English language could not be held there. It is somewhat paradoxical to have to wait for an era of ever increasing core memory for Bloom filters to broadly take hold.

Simply stated, a Bloom filter achieves space efficiency (less than $O(mn)$ space) by allowing for a small probability of false positives, but no false negatives, to the set-membership question.

We want to test for membership in a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements. The universe U of elements is typically very large. Let $h_1, \ldots, h_k$ be k independent hash functions with range $\{1, \ldots, m\}$. The following algorithm is a Bloom filter.

```
MakeDictionary
   Array $B[1 \ldots m]$
   Initialize $B[j] = 0$ for $j = 1, \ldots, m$

Insert($x$)
   For $i = 1, \ldots, k$
      $B[h_i(x)] = 1$
   Endfor

Lookup($x$)
   For $i = 1, \ldots, k$
      If $B[h_i(x)] = 0$ then
         Return false
   Endfor
   Return true
```

To add an element $x \in S$, the bits $h_i(x)$ in the array B are set to 1 for $1 \leq i \leq k$. To check whether an element x is in S, we check whether all the bits $h_i(x)$ in the array B are set to 1. If they are not, then x clearly cannot be an element of S. If they are, we declare that x is an element of S, even though this might not be true. We now calculate the probability that x is not a member of S, i.e. the probability of a false positive.

The hash functions are completely random, therefore the probability that a bit in B (say bit j) is still 0 after all n elements of S are added is

$$\begin{aligned} \Pr(B[j]=0) &= \left[\Pr(\text{some } x \in S \text{ left } B[j]=0)\right]^n \\ &= \left[\left[\Pr(\text{some } h_i(x) \text{ left } B[j]=0)\right]^k\right]^n \\ &= \left[\left(1-\Pr(\text{some } h_i(x) \text{ set } B[j]=1)\right)^k\right]^n \\ &= \left(1-\frac{1}{m}\right)^{kn} \\ &\approx e^{-kn/m} \end{aligned} \tag{1}$$

Now, the probability of a false positive is the probability that all k bits $h_i(x)$ are set. Therefore,

$$\begin{aligned} \Pr(\text{false positive}) &= \left[\Pr(B[j]=1)\right]^k \\ &= \left[1-\Pr(B[j]=0)\right]^k \\ &\approx \left(1-e^{-kn/m}\right)^k \end{aligned} \tag{2}$$

We define

$$\begin{aligned} p &= e^{-kn/m} \\ f &= (1-p)^k \end{aligned} \tag{3}$$

Clearly, the probability f of a false positive (or *false positive rate*) decreases as m/n gets larger and as the number k of hash functions gets larger. It is natural to ask ourselves what the optimal k is, given m and n. Now, from Eq. (3), it follows immediately that

$$\ln f = -\frac{m}{n} \ln p \ln(1-p) \tag{4}$$

The global minimum of $\ln f$ (and therefore of f), by symmetry, is at $p = 1/2$. Therefore, the optimum we seek is

$$\begin{aligned} k &= \frac{m}{n} \ln 2 \\ f &= \left(\frac{1}{2}\right)^k = (0.6185)^{m/n} \end{aligned} \tag{5}$$

Eq. (5) provides a very useful relation between the number of storage bits per element, m/n, and the false positive rate f. Achieving a desired false positive rate f fixes m/n.

5 Space and time complexity of the Bloom filter

Assuming memory usage scales linearly with the number of elements:

$$m = cn \tag{6}$$

we get the obvious fact that the memory usage of a Bloom filter is $O(cn)$. This compares very favorably to a standard, non-probabilistic, data structure such as a balanced binary search tree, which uses $O(mn)$ space to store n elements that are m bits long. As we just saw, the parameter c (bits per element) is an important one, and determines the false positive rate.

The running time of the Bloom filter `insert` and `lookup` algorithms is the time it takes to compute the k hash values. Therefore, the running times of an insertion and lookup are $O(c)$. By comparison, the insertion and lookup times for a balanced binary search tree is $O(\log n)$.

Note that the running time increases as the false positive rate increases, since more hash values need to be computed. Pagh, Pagh and Rao propose in [5] an improvement that runs in $O(1)$ time, independent of the false positive rate.

6 Properties of the Bloom filter

Two useful properties of the Bloom filter for access control are:

1. The union of two Bloom filters of the same size and using the same hash functions can be obtained by bitwise ORing the two filters.
2. The intersection of two Bloom filters of the same size and using the same hash functions can be obtained by bitwise ANDing the two filters.

For example, when two groups of users are merged into one, their combined ACL can be obtained with minimal computation.

7 Counting filters

There is no `delete` operation in the standard Bloom filter. This is because each 1 bit in the filter B could have been set by multiple elements, therefore it cannot simply be unset when we attempt to delete an element. To get around this problem, counting filters are used [6].

In a counting filter, B, instead of being an array of m bits, is an array of m counters (each of which takes up a fixed number of bits, say four). When an element is added to the counting filter, instead of setting bits in B, `insert` increments the counters corresponding to the k hash values. When an element is deleted from the counting filter, one decrements the counters corresponding to the k hash values. In this way, one gets a data structure that is a true *dictionary* [7].

In the case of access control, modifications to the ACLs occur infrequently enough that one can typically regenerate Bloom filters from scratch.

8 Bloomier filters

A generalization of a Bloom filter can be used to construct a *probabilistic* associative array called a Bloomier filter [8].

In its simplest form, the Bloomier filter is a cascade of Bloom filters. Let us look at how to represent a single bit in the associative array, as an example.

Let the Bloom filter A_0 contain all keys mapping to 0 and the Bloom filter B_0 contain all the keys mapping to 1. During lookup, we inspect both A_0 and B_0. If the key is in A_0 and not in B_0, then there is a high probability that the value associated with the key is 0. If it is in B_0 and not in A_0, then the value is probably 1. When *both* A_0 and B_0 say that they contain the key, at least one of them is wrong, and we have a *false positive*; to resolve this, we use a second set of bloom filters A_1 and B_1.

The Bloom filter A_1 contains all the keys that map to 0 and are false positives in B_0, and B_1 contains all the keys that map to 1 and are false positives in A_1. Again, it may happen that both A_1 and B_1 declare that they contain a given key, in which case we need a new set of filters A_2 and B_2.

In general, we construct α sets of filters A_i and B_i, $i = 0, \ldots, \alpha$ such that (using the notation in [8]):

$$\begin{aligned} A_i &= A_{i-1} \cap B^*_{i-1} \\ B_i &= B_{i-1} \cap A^*_{i-1} \end{aligned} \tag{7}$$

where A_i^* and B_i^* are the set of false positives in A_i and in B_i respectively.

Each successive pair A_{i+1}, B_{i+1} is much smaller in size than the previous pair A_i, B_i. It is shown in [8] that the storage is still $O(cn)$ bits with high probability, and the search time is $O(\log \log n)$ in the worst case but $O(1)$ on average.

9 Deny ACLs

It is now clear how one would implement a probabilistic filter for a Deny ACL. One simply inserts all the URLs from the Deny ACL into a Bloom filter. There are no false negatives, so if our Bloom filters declares that an incoming URL is not in the Deny ACL, this is guaranteed to be true and we can let that request go through. If on the other hand, we get a hit, we use our slow-but-certain access check to determine if the URL in question is indeed part of the Deny ACL.

Depending on the desired false positive rate, set the number of bits per entry in the Bloom filter using Eq. (5). For example, to achieve a 1% false-positive rate, one needs $m/n = 9.6$ bits per entry.

If the majority of the URLs requested by users are not in the Deny ACL (as is typically the case in a corporate environment), using a Bloom filter can result in a huge improvement in throughput.

10 Permit ACLs

It is unclear how one might use a probabilistic filter to speed up checks against a Permit ACL. This is because, unlike in the previous case, we would like to achieve *zero false positives* with a certain probability of a false negative.

Let us see how one might use a Bloomier filter, as an exercise. The keys of this probabilistic associative array are the URLs in the Permit ACL. The values associated with those keys are cryptographically-safe hashes (e.g. SHA-256 hashes) of the URLs. A lookup then consists of making sure that the incoming URL corresponds to the value obtained from the Bloomier filter. While this method could possibly take up *more* space than a deterministic algorithm, it does have the advantage of fast lookups ($O(c)$ on average, as noted above). This is especially true if one exploits the inherent parallelism in the calculation of the k hashes.

11 Open question

A probabilistic algorithm that produces *zero false positives*, even with a large probability of a false negative, would be very useful for the Permit ACL case. The important characteristics of such an algorithm would be space-efficiency and fast lookup times. Addition and deletion of elements are rare events that need not be optimized. In this case too, the universe U of URLs is much bigger than the set S of URLs in the Permit ACL.

Whether one can design such an algorithm is an open question.

12 Conclusion

Bloom filters can play an important role even in applications where error is not tolerated, such as in network security. Using them to filter out the bulk of the traffic, so that a slower deterministic policy engine is not inundated, is an effective strategy to keep up with ever increasing line speeds. Two properties of the Bloom filter stand out in particular. The first, space efficiency, is obvious. The second is the inherent parallelism in the lookup; all k hash computations can be done in parallel [3].

References

[1] Burton H. Bloom, *Space/Time trade-offs in hash coding with allowable errors*, Communications of the ACM, vol. 13, no. 7, 1970.

[2] Andrei Broder and Michael Mitzenmacher, *Network Applications of Bloom Filters: A Survey*, Allerton 2002.

[3] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, John W. Lockwood, *Deep Packet Inspection using Parallel Bloom Filters*, IEEE Micro, vol. 24, no. 1, pp. 52-61, 2004.

[4] Elliott Karpilovsky, *Bloom Filters*, slides, 2005.

[5] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao, *An Optimal Bloom Filter Replacement*, Proceedings of the Sixteenth ACM-SIAM symposium on discrete algorithms, pp. 823-829, 2005.

[6] Li Fan, Pei Cao, Jussara Almeida and Andrei Z. Broder, *Summary cache: a scalable wide-area web cache sharing protocol*, IEEE/ACM Transactions on Networking, vol. 8, no. 3, pp. 281-293, 2000.

[7] Jon Kleinberg, Éva Tardos, *Algorithm Design*, Addison-Wesley, 2005.

[8] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal, *The Bloomier filter: an efficient data structure for static support lookup tables*, Proceedings of the Fifteenth Annual ACM-SIAM symposium on discrete algorithms, pp. 30-39, 2004.