**Space efficient storage of mails for Email Monitoring**

**Abstract**

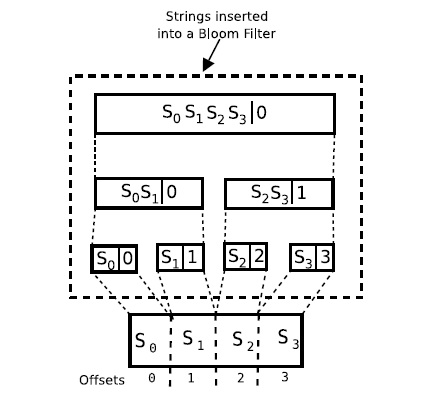
Given an excerpt of body of the mail, finding whether a mail with that content has passed through the mail server or not.

**Capturing Traffic:**

We installed send mail server and captured the emails that passed through the mail server using milter API library.

We then extracted sender, receiver and body of the mail that will be used for further forensic analysis. Storing entire body of the mail is not feasible, as the amount of storage required is huge. So, we used a data structure called a Hierarchical Bloom Filter (HBF). An HBF creates compact digests of payloads and provides probabilistic answers to membership queries on the excerpts of payloads.

**Implementation of Hierarchical Bloom Filter:**



**Inserting string “S0S1S2S3” into a Hierarchical Bloom Filter.**

Implementing the hierarchy using a single Bloom filter with the offset of each element concatenated to it during insertion, like (content||offset), improves the space utilization. For example, in order to store string “S0S1S2S3” in the hierarchy, we need to insert the following strings into the Bloom filter {(S0S1S2S3||0), (S0S1||0), (S2S3||1), (S0||0), (S1||1), (S2||2), (S3||3)}. Having a single Bloom filter allows us to maximize its space utilization as we can determine the optimal number of elements inserted into it a priori.

When saturation point is reached, we use new bloom filter to store the contents of mail. When a Huge mail is captured and saturation point occurs in the middle of the mail, then what to do?

If saturation point does not occur, write bloom filter contents to file. If saturation point occurs, we don’t write bloom filter contents to that file, we open new file put those contents into file.

We implemented Logging based on timestamp factor.

**Saturation Point:**

A Bloom filter is a simple, space-efficient, randomized data structure for representing a set in order to support membership queries. It uses a set of k hash functions of range m and a bit vector of length m. Initially, the bit vector is set to 0. An element in the set is inserted into the

Bloom filter by hashing the element using the k hash functions and setting the corresponding bits in the bit vector to 1. To test whether an element was inserted into the filter, we simply hash the element with the same hash functions and if all corresponding bits are set to 1 then the element

is said to be present in the filter. The space efficiency of a Bloom filter is achieved at the cost of a small probability of false positives as defined by Equation, where n is the number of elements in the set

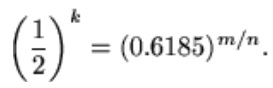
Untitled.png

K – Number of hash functions.

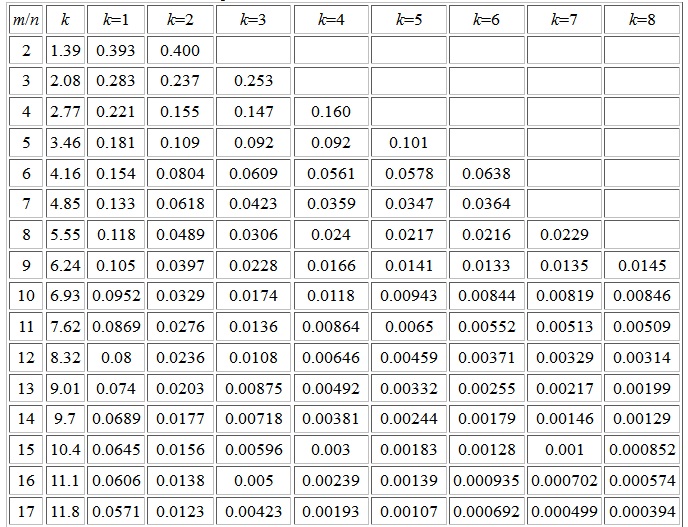
m – Bit vector length.

n – Number of elements in the set.

The right hand side is minimized for K = ln 2 × m/n , in which case it becomes



The false positive ratios for common combinations of *m*/*n* and *k* are given below.



From the above table,we chose k=4 and m/n factor to be 15,so false positive rate becomes

0.003.We chosen this parameters to make false positive rate as minimum as possible.We chosen bit vector length m=65536.

Known Parameters are

K = 4

m = 65536

n = ?

Substituting above parameters in this equation K = ln 2 × m/n, we got n=11356.So,Saturation Point becomes n=11356 (number of blocks that can be inserted into bloom filter).we chose block size to be 32 bytes.

**Querying:**

The hierarchical nature of the HBF resolves collisions automatically. Suppose we would like to verify if we have actually seen a string of the form “S0S1S2 S3”. As in BBF, the string is broken down into four individual query strings {S0, S1, S2, S3}. By trying all possible offsets at the bottom of the hierarchy we can verify the existence of strings {(S0 || i), (S1 || i + 1), (S2 || i + 2),

(S3 || i + 3) with false positive rate FP. Since ‘S0’ and ‘S1’ are subsequent in the query string we can improve the confidence of the results by verifying query string (S0S1||i) at the level above. In this way we can verify the matches at different levels of the hierarchy. For example, we can verify the whole string “S0S1S2S3” all the way to the top of the hierarchy consequently improving the confidence of the result at each level.