

Payload Inspection Using Parallel Bloom Filter in Dual Core Processor

Arulanand Natarajan (Corresponding author)

Anna University Coimbatore, TN, India

E-mail: arulnat@yahoo.com

S. Subramanian

Sri Krishna College of Engineering and Technology, TN, India

E-mail: dsraju49@gmail.com

Abstract

This paper presents payload inspection for identification of spam files using bloom filter in dual core processor. Spam files flood the Internet in an attempt to dump the messages on recipients who do not intend to receive it. Spam costs the sender very little to send and most of the costs are levied to the recipients or the carriers. The proposed system identifies and filters the incoming spam files using Bloom filter algorithm implemented in dual core processor. The results of the Bloom filter algorithm are examined and these results demonstrate the performance of Sequential Bloom filter and Parallel Bloom filter in a Dual Core Processor.

Keywords: Bloom filter, Dual Core Processor, Spam, Payload Inspection

1. Introduction

An electronic message is a spam if the recipient's personal identity and context are irrelevant because the message is equally applicable to many other potential recipients and recipient has not granted explicit permission for it to be sent. The transmission and reception of the message appears to the recipient to give a disproportionate benefit to the sender. Most spam is commercial advertising, often for dubious products, get-rich-quick schemes, or quasi-legal services. Spaming remains economically viable because advertisers have no operating costs beyond the management of their mailing lists and it is difficult to hold senders accountable for their mass mailings.

Payload is the essential data that is being carried within a packet or other transmission unit. The payload does not include the overhead data required to get the packet to its destination. In general usage, the payload consists of the bits that get delivered to the end user. Bloom filters are compact data structures for probabilistic representation of a set to support membership queries. Its core concept is associative containers. Given a string X, the Bloom filter computes k hash functions on it producing k hash values ranging from 1 to m. It then sets k bits in an m-bit long vector at the location corresponding to k hash values. This procedure is repeated for all the members of the set. This process is called programming the bloom filter. The query process is similar to programming the filter. A string whose membership is to be verified is sent as input to the bloom filter algorithm to generate k hash values using the same hash functions that were used to program the filter. The bits in the m-bit long vector at the locations corresponding to the k hash values are looked up. If at least one of these k bits is not found in the m-bit long vector, then the string is declared to be a nonmember of the set. However, if all the bits are found in the m-bit long vector, then the string is said to belong to the set with a certain probability. This uncertainty in the membership comes from the fact that those k bits in the m-bit vector can be set by any other n-1 members. Thus finding a bit set does not necessarily imply that it was set by the particular string being queried. However, finding a bit not set certainly implies that the string does not belong to the set.

In a single-core processor, the CPU is fed with strings of instructions. The CPU executes the instructions and then selectively stores it in its cache for quick retrieval. When data that does not reside on the cache is required, it is retrieved through the system bus from RAM or from storage devices. Accessing data outside the cache slows down the performance of the CPU. In a dual core processor, each core handles incoming data strings simultaneously using multi-threading to improve efficiency. When one core is executing, the other core can either access the system bus or execute its own code.

This paper deals with the performance of the Dual core processor for payload inspection using multi-threading concept. Section 2 provides the methods of spam filters. Section 3 gives a general overview of the Bloom filter. The proposed system is explained in Section 4. The experimental setup and performance analysis of the results are described in Section 5.

2. Spam Filters

This section describes several methods of spam filtering. The complexity, advantages and limitations of each method are explained.

2.1 Rule-based Filtering

In a rule-based algorithm, rules are defined to classify emails as spam based on the characteristics of the message. As an example, a rule could be that all emails with magenta-colored text are spam. A good rule-based filter combines all the rules to make a decision. The problem with rule-based filters is that the rules are written looking for the obvious characteristics of spam. Spams are written in such a way that it appears like a normal message. As a result, it is hard to write simple rules that work well in all cases.

2.2 Blacklisting

Blacklisting is a form of rule-based filtering that uses one rule to decide which email messages are spams. A blacklist is a list of traits that spam emails have, and if an email being tested contains any of those traits, it is marked as spam. It is possible to organize a blacklist based on “From:” fields, originating IP addresses, the subject or body of the message, or any other part of the message that makes sense.

Blacklists can be used either at a large scale or a small scale. A large-scale blacklist would usually be provided by a third party. The user typically does not contribute to a large list. On a smaller scale, the user could simply tell their email client not to allow email from certain addresses. A small-scale blacklist works fine if the user gets spam from one particular address. On a larger scale, where the user does not have any control over the blacklist, there must be a mechanism in place for dealing with accidental blacklisting of other users.

2.3 Whitelisting

While blacklisting is a way of deciding emails which are spam, whitelisting decides which emails are non-spam and assumes that the rest of the messages are spam. Users would presumably whitelist everyone that they would expect to receive email from. The obvious problem is that it is impossible to predict who is going to send email, and anyone previously unknown to the user will be filtered out. One way to avoid this problem is to read through the filtered email regularly.

Another method is to let the senders of all emails to be marked as spam, while providing them with a method of getting added to the whitelist. It probably blocks all spam, but there is still a problem of dealing with automated order confirmations and mailing lists. When a user joins the mailing list, that address can be included in their whitelist, but it has to be done manually. For order confirmations, for example, the user can not always know from where the confirmation will be coming. Hence this approach is also subjected to some limitations.

2.4 Paul Graham's Bayesian Filtering

Bayesian filters use probabilistic reasoning to decide whether or not a message is spam. These filters work on Bayes' rule, which is useful for calculating the probability of one event when one knows another event is true. In the case of email, the rule is used to determine the probability that an email is spam given that it contains certain words.

The probability that an email is spam is based on the words it contains. The filter needs to know about the emails that a user receives. Since the interest is solely in the words and their frequencies (and not their ordering in this implementation), the solution is to keep a hash table to record how often each word appears. Spams and non-spams are kept in a separate hash table, so that the probabilities can be calculated later. When an email is declared spam, the spam table is updated by incrementing the frequency count of each word contained in that email. Similarly, non-spam counts are incremented. Over a period of time, the hash tables begin to characterize a person's spam and non-spam messages.

Graham (2003) suggests a modified Bayes' rule to calculate probabilities. Bayes' rule combines multiple probabilities:

$$P(A|B \wedge C) = \frac{P(A|C) P(B|A \wedge C)}{P(B|C)} \quad (1)$$

Graham's modification of Baye s' rule is:

$$P(A|B \wedge C) = \frac{P(A|C) P(A|C)}{P(A|B) p(A|C) + (1-P(A|B)) (1-P(A|C))} \quad (2)$$

$P(A|B \wedge C)$ is the probability that event A is true, given that events B and C are true. In the case of spam filtering, event A would represent an email being spam, while B and C correspond to certain words being in the email.

Probability that an email is spam (A) is known when it contains words B and C. This equation can be expanded to include more words (Graham proposes using the most interesting 15 words). Graham's Bayes' rule assumes that one already knows the probability that an email is spam for each individual word. This is calculated with the following formula:

$$P(\text{Spam}|\text{word}) = \frac{\frac{b}{nbad}}{\frac{g}{ngood} + \frac{b}{nbad}} \quad (3)$$

Here 'b' and 'g' represent the number of times a word appears in spam and non-spam emails respectively and 'nbad' and 'ngood' represent the total numbers of spam and non-spam emails received respectively.

3. Bloom filter

Bloom filter implements the above concepts through the use of multiple distinct hash functions. Bloom filter guarantees that for any membership query there will never be any false negatives; however there may be false positives. The false positive probability can be controlled by varying the size of the table used for the Bloom filter and also by varying the number of hash functions.

Subsequent research work done in the area of hash functions and tables on Bloom filters by Border and Mitzenmacher (2004), suggests that an optimal Bloom filter (one which provides the lowest false positive probability for a given table size) constructed with at most two distinct hash functions, greatly increases the efficiency of membership queries. Bloom filters are commonly found in applications such as spell-checkers, string matching algorithms, network packet analysis tools and network/internet caches.

3.1 Literature Review

In figure 1 the arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set {x, y, z}, because it hashes to one bit-array position containing 0. In this figure 1, the values of m and k are taken as 18 and 3 respectively.

An empty Bloom filter is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions with a uniform random distribution. To add an element, each of the k hash functions is fed to get k array positions. The bits at all these positions are set to 1.

To query for an element, each of the k hash functions is fed to get k array positions. If any of the bits at these positions are 0, the element is not in the set. Otherwise all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements.

A Bloom filter program consists of a set of hash functions, a hash function buffer to store hash results temporarily, a look up array to signify hash values and a decision component made of an AND to test the membership of testing string as shown in figure 2.

The requirement of designing k different independent hash functions may be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple different hash functions by slicing its output into multiple bit fields. Alternatively, pass k different initial values (such as 0, 1, 2, ..., k-1) to a hash function that takes an initial value; or add these values to the key. For larger m and/or k, independence among the hash functions can be relaxed with negligible increase in false positive rate (Dillinger & Manolios 2004a; Kirsch & Mitzenmacher 2006). Specifically, Dillinger & Manolios (2004b) show the effectiveness of using enhanced double hashing or triple hashing, variants of double hashing, to derive the k indices using simple arithmetic on two or three indices computed with independent hash functions.

Unfortunately, removing an element from this simple Bloom filter is not possible. The element maps to k bits, and although setting any one of these k bits to zero suffices to remove it. This has the side effect of removing any other element that maps onto that bit and there is no way of determining whether any such element has been added. Such removal may introduce a possibility for false negatives, which is not allowed.

Removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which is not permitted. This approach also limits the semantics of removal since re-adding a previously removed item is not possible. However, it is often the case that all the keys are available but are cumbersome to

enumerate. When the false positive rate goes very high, the filter can be regenerated; this should be a relatively rare event.

Bloom filters can be implemented in hardware but they have their own limitations. The hardware Bloom Filters are expensive. They are primarily used in huge mail servers. The initial data set is fixed in these hardware filters and updating these cost time and money. The Hardware systems are also prone to frequent failures and they have to be shut down when they are being updated thus leaving the network vulnerable. The software implementation proposed here is primarily aimed at small scale levels where hardware implementation is not a viable option. The proposed parallel implementation boosts performance in the long run. Dharmapurikar et al (2004) have presented a hardware-based technique using Bloom filters, which can detect strings in streaming data without degrading network throughput. They group signatures according to their length (in bytes) and store each group of string in a unique Bloom filter. An analyzer is employed to resolve false positives. They have also proposed a technique for reducing packet inspection time by using parallel Bloom filters.

Artan and Chao (2005) have proposed a space-efficient method to follow and detect signatures that are fragmented over multiple packets. Prefix Bloom filter was used along with chain heuristic to achieve this purpose. A fault in Bloom filters, however, may cause false negatives to occur. For a string already programmed in a Bloom filter, a faulty hashing unit might generate an incorrect location (i.e., a hash value) at which 0 is stored instead of 1, resulting in a false negative. For a given fault, the probability that false negatives will occur is high unless some provisions are made to detect and eliminate them.

Myeong-Hyeon Lee and Yoon-Hwa Choi (2007) presented a hardware-based fault-tolerant Bloom filter which detects and eliminates false-negatives during normal operation. It is based on property checking of a Bloom filter with some extra hardware circuits. The design is simple to implement with negligible overhead. As a result, packets may proceed at line speed, regardless of the added circuits.

3.2 Space and Time Complexity

Bloom filter is used to speed up answers in a key-value storage system. Values are stored on a disk which has slow access time. Bloom filter decisions are much faster. However some unnecessary disk accesses are made when the filter reports a false positive. Overall answer speed is better with the Bloom filter than without the Bloom filter. Use of a Bloom filter for this purpose, however, does increase memory usage.

While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked list of the entries.

Most of these require storage space for at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). Linked structures incur an additional linear space overhead for pointers.

A Bloom filter with 1% error and an optimal value of k , on the other hand, requires only about 9.6 bits per element regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. If 1% false positive rate seems to be too high, then each time about 4.8 bits per element are added, it decreases it by ten times.

Bloom filters also have the unusual property that the time needed to either add items or to check whether an item is in the set is a fixed constant, $O(k)$, completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, the Bloom filter works better because its k lookups are independent and can be parallelized.

To understand space efficiency, it is necessary to compare the general Bloom filter with its special case when $k = 1$. If $k = 1$, then in order to keep the false positive rate sufficiently low, a small fraction of bits should be set, which means the array must be very large and contain long runs of zeros. The information content of the array relative to its size is low. The generalized Bloom filter (k greater than 1) allows many more bits to be set while still maintaining a low false positive rate, when the parameters (k and m) are chosen well.

3.3 False Positive Rate

The false positive probability p is defined as a function of number of elements n in the filter and the filter size m . An optimal number of hash functions $k = (m / n) \ln 2$ has been assumed. A hash function selects each array position with equal probability. If m is the number of bits in the array, the probability that a certain bit is not set

to one by a certain hash function during the insertion of an element is given by

$$1 - \frac{1}{m} \quad (4)$$

The probability that it is not set by any of the hash functions is

$$\left(1 - \frac{1}{m}\right)^k \quad (5)$$

If n elements are inserted, the probability that a certain bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \quad (6)$$

The probability that it is 1 is therefore

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \quad (7)$$

The test membership of an element is not in the set. Each of the k array positions computed by the hash functions is 1 with a probability as given above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is represented as

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \quad (8)$$

The probability of false positives decreases as m (the number of bits in the array) increases, and increases as n (the number of inserted elements) increases. For a given m and n , the value of k (the number of hash functions) that minimizes the probability is

$$\frac{m}{n} \ln 2 \approx \frac{9m}{13n} \approx 0.7 \frac{m}{n} \quad (9)$$

which gives the false positive probability of

$$2^{-k} \approx 0.6185^{\frac{m}{n}}$$

By taking the optimal number of hashes, the false positive probability (when ≤ 0.5) can be rewritten and bounded (Starobinski et al 2003) as

$$\frac{m}{n} \geq \frac{1}{\ln 2} \quad (10)$$

In order to maintain a fixed false positive probability, the length of a Bloom filter must grow linearly with the number of elements being filtered. The required number of bits m for given n (the number of inserted elements) and a desired false positive probability p , and assuming the optimal value of k is

$$m = -\frac{n \ln p}{(\ln 2)^2} \quad (11)$$

Table 1 shows the false positive ratios for common combinations of m/n and k in Bloom Filter

4. Parallel Bloom Filter in a Dual Core Processor

Two basic operations are defined for Bloom Filter. The first one is programming the look up array using hash functions for all the strings in the data set. The second one is for checking the membership of a test string. Bloom Filter represents the set of n -signatures $X = \{X_1, X_2, \dots, X_n\}$ in an m -bit array. The elements in this array are set to '0' before programming. Each signature is of b bits and is hashed k -times by independent hash functions $H_1, H_2, H_3, \dots, H_k$. It is assumed that each hash function maps uniformly to a random number in range $\{0, 1, 2, \dots, m-1\}$ where m defines the number of bits in look up array. The random number describing hash function value indicates a bit location in m -bit look up array, which is then set to '1'. A particular bit location in m -bit look up array can be set to 1 more than once.

In the testing phase, a string is tested for membership by querying the programmed Bloom Filter. The string is hashed k -times as before. If all the hash values point to the bit locations that are set to '1' then this indicates that the test string may be a member of the set with a certain probability (false positive) which is called as match. If any one of the hash values points to a bit location that is set to '0' then the test string is definitely not a member of the set and is called as mismatch.

5. Experiment Results

Hash functions with high distribution coefficients are chosen to avoid collisions. A hash function is said to have a high distribution coefficient, if the signatures generated by the hash function have a wide range. This is essential for hash functions chosen for designing Bloom filters as they operate on a large dataset. Wide distribution of the hash function reduces the probability of collisions. To implement this system in the proposed work, two stable hash functions are chosen with high distribution coefficient, namely the AP hash function and the BKDR hash function. The initial dataset is populated with spam strings acquired from cooperative spam databases which contain the most widely countered spam strings.

The bit vector is of a predefined size which is calculated with respect to the size of the initial dataset and the allowable probability of false positives. In this work, 210 words are taken as spam words and the length of the bit vector is set as 2048. The false positive ratio for the proposed work for given m, n and k is 0.0329 which is shown in the Table 1. The signatures generated by the hash functions are integers and they are normalized to the size of the bit vector and their position is mapped onto the bit vector. The corresponding position on the bit vector is set to 1. The same position in the bit vector can be set to 1 multiple times. Thus the signatures generated by two hash functions for all the spam strings are mapped on to the bit vector.

For experiment results, 10 different 1 KB spam files are chosen for testing. The strings are parsed from the payload and compared with the bit vector generated using the initial dataset. The strings are tested for membership. If a string parsed from the user file passes the membership test then the string is present in the initial dataset and the file is classified as spam. In sequential membership, testing method employs a single membership testing function for the parsed strings of all length. The normalized signatures generated for a string in the initialization phase are fed into this function. The normalized signatures denote positions in the Bit Vector. If the positions denoted by the normalized signatures of a string in the Bit Vector are all set to 1, then the string passes the membership test. Even if a single position is set to 0, then the string fails the membership test. In Parallel membership, there are multiple testing functions each dedicated to strings of particular length. The operating principle is same as that of the sequential membership testing function. But here there are multiple independent sequential membership testing functions that can be executed in parallel. Figure 4 shows the experiment results obtained from sequential and parallel bloom filters in dual core processor with 1 KB spam files. The execution time of both sequential and parallel bloom filter depends on the spam word position of the payloads.

The performance of the parallel bloom filter outperforms the sequential bloom filter in a dual core processor. In a dual core processor, the parsing of a string from the payload, hash key generation and the query matching process are all done in parallel for different size of strings using multithreading concept. The main advantages of multi threading is that, if a thread gets couple of cache misses, the other thread(s) can continue running, taking advantage of the unused computing resources. This leads to faster overall execution, as these resources would have been idle if only one single thread was executed. Also, if several threads work on the same set of data, they can actually share their cache, leading to better cache usage and synchronization on its values. The advantages of dual core processor is that the computer will use less energy and delivers better performance from both its cores compared to a single high performance chip design.

Conclusion

In this work, both sequential and parallel Bloom filters are implemented for payload inspection in a dual-core processor. A dual core processor has the advantage of boosting the system's multithreading computing power. It improves the utilization of the operating system and the applications running on the computer that supports thread-level parallelism. As the experimental results show, parallel bloom filters implemented in a dual core processor has the potential to increase the execution speed of bloom filters using software implementation. Bloom filter algorithm can be easily extended and adapted for parallel execution using multi-cores. With today's availability of dual core, quad core and 8 core processors this speed could supersede the speed that can be achieved through hardware implementation. This increase in speed can be used advantageously in many applications. Hence implementation of parallel bloom filters using multi core processor based on multithreading concept provides a large scope for further investigation and research.

References

- Artan, N. S., & Chao, H. J. (2005). Multi-packet Signature Detection using Prefix Bloom Filters. *Proceedings of 48th Annual IEEE Global Communications Conference*.
- Bloom, B. (1970). Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 422–426.

- Border, A., & Mitzenmacher, M. (2004). Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4), 485–509.
- Graham, P. (2003). Better Bayesian Filtering. *Proceedings of Spam Conference*.
- Dharmapurikar, S., Krishnamurthy, P., Sproull, T. S., & Lockwood, J. W. (2004). Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 52–61.
- Dillinger, P., C., Manolios, P. (2004a). Fast and Accurate Bitstate Verification for SPIN, *Proceedings of the 11th International Spin Workshop on Model Checking Software*, Springer-Verlag, Lecture Notes in Computer Science 2989.
- Dillinger, P., C., & Manolios, P. (2004b). Bloom Filters in Probabilistic Verification, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*, Springer-Verlag, Lecture Notes in Computer Science 3312.
- Kirsch, A., & Mitzenmacher, M. (2008). Less Hashing, Same Performance: Building a Better Bloom Filter, *Random Structures & Algorithms*, 33(2), 456–467.
- Lee, M., & Choi, Y. (2007). A Fault-Tolerant Bloom Filter for Deep Packet Inspection, *Proceedings of 13th Pacific Rim International Symposium on Dependable Computing*, 389–396.
- Starobinski, D., Trachtenberg, A., & Agarwal, S. (2003). Efficient PDA Synchronization, *IEEE Transactions on Mobile Computing*, 2(1), 40.

Table 1. False Positive Ratio

m/n	K	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846
11	7.62	0.0689	0.0276	0.0136	0.00864	0.0065	0.00552	0.00513	0.00509
12	8.32	0.08	0.0236	0.0108	0.00646	0.00459	0.00371	0.00329	0.00314
13	9.01	0.074	0.0203	0.00875	0.00492	0.00332	0.00255	0.00217	0.00199
14	9.7	0.0689	0.0177	0.00718	0.00381	0.00244	0.00179	0.00146	0.00129
15	10.4	0.0645	0.0156	0.00596	0.003	0.00183	0.00128	0.001	0.000852
etc...									

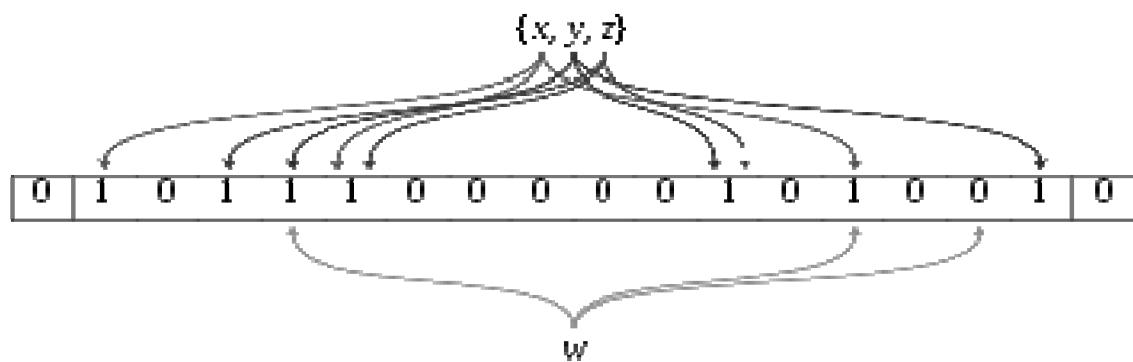


Figure 1. An Example of Bloom filter representing the set {x,y,z}

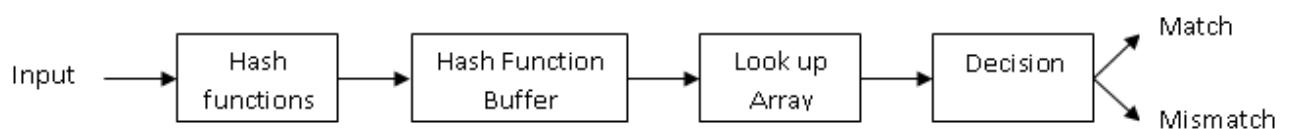


Figure 2. Block diagram for checking membership of a string.

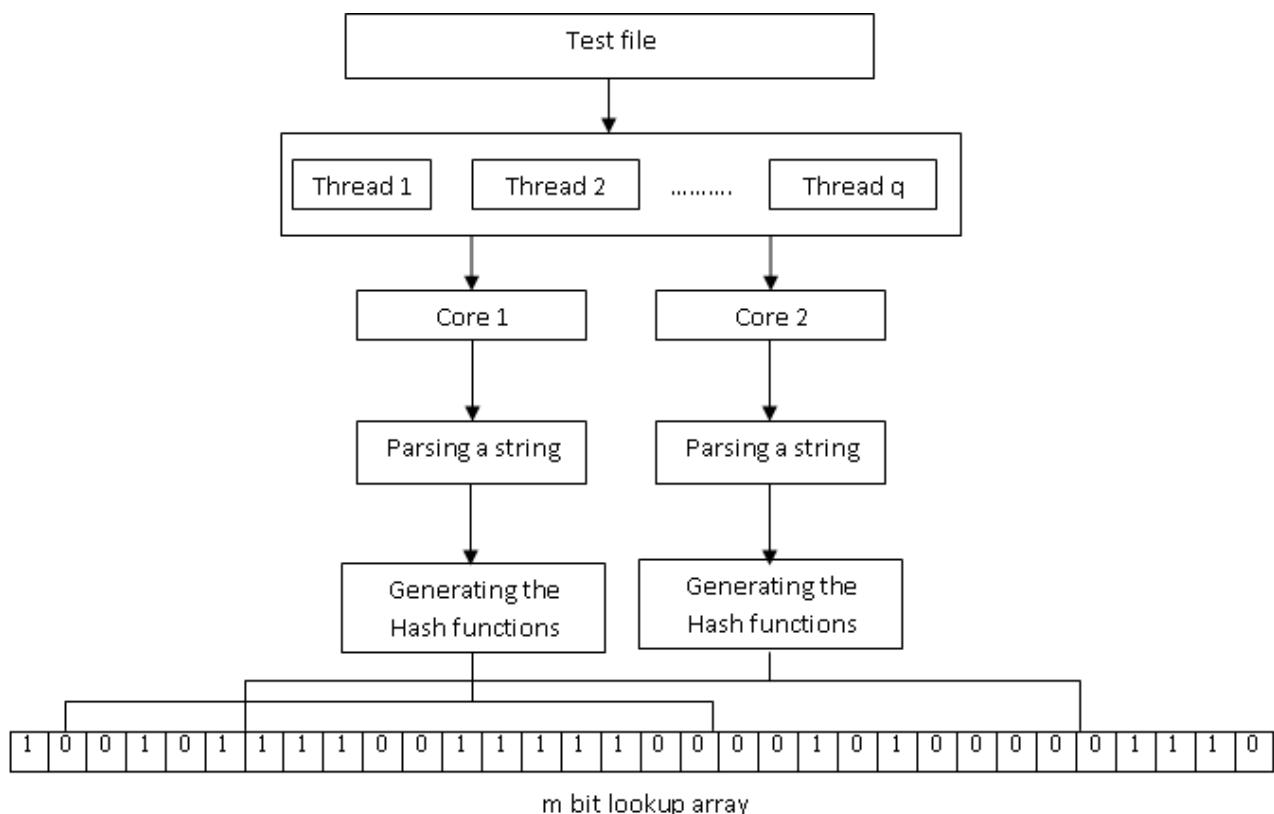


Figure 3. Parallel Bloom Filter for Payload Inspection

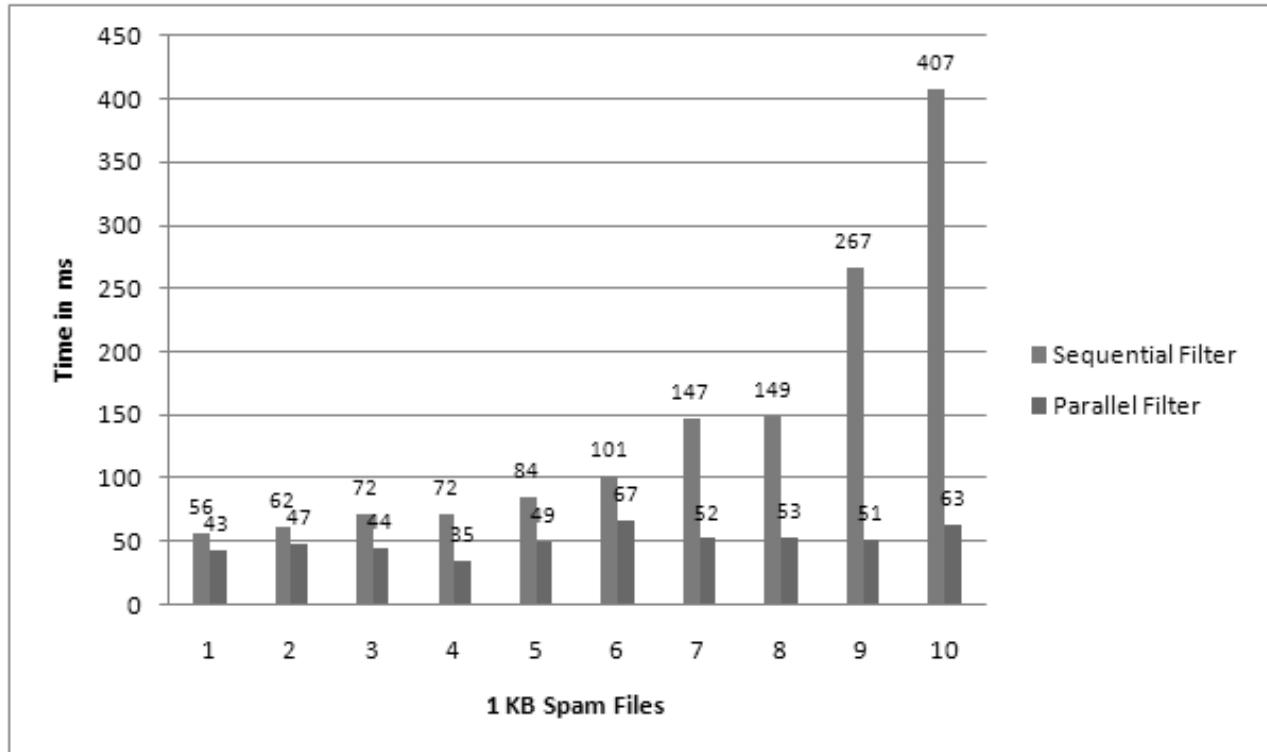


Figure 4. Experiment results using sequential and parallel bloom filter