Aggregated Bloom Filters For Intrusion Detection And Prevention Hardware

N. Sertac Artan, Kaustubh Sinkar, Jalpa Patel, and H. Jonathan Chao
Department of Electrical and Computer Engineering
Polytechnic University, Brooklyn, NY

Abstract—Bloom Filters (BFs) are fundamental building blocks in various network security applications, where packets from high-speed links are processed using state-of-the-art hardware-based systems. In this paper, we propose Aggregated Bloom Filters (ABFs) to increase the throughput and scalability of BFs. The proposed ABF has two methods to improve average speed and scalability. The first method leverages the query mechanism for hardware BFs. We optimize queries by removing redundant hash calculations and memory accesses. First, to remove redundancy, the hash functions for each query are calculated sequentially. As soon as we have a no match in any of the hash results, the query is immediately abandoned. We then aggregate multiple queries and query a BF with all of these queries in parallel, which maximizes the throughput of the BF. The second method addresses scalability issues regarding the on-chip memory resources. In most applications multiple BFs are required to store many sets with different numbers of elements. These sets may also be too small for the unit memory on-chip. So, most of the memory is left unused, causing low memory utilization. The second method aggregates small distributed BFs to a single BF allowing better on-chip memory utilization. For the application of Network Intrusion Detection and Prevention Systems (NIDPSs), our proposed ABF shows seven-fold improvement in the average query throughput and four times less memory usage.

I. INTRODUCTION

Bloom Filters (BFs) [1] are highly popular data structures used in many applications. In particular, BFs are used extensively for network security applications such as network intrusion detection and prevention [2], [3], virus scanning [4], worm detection [5], [6], Denial-of-Service prevention [7], and network forensics [8]. Most of the time, attack traffic should be sifted from data on very high-speed links. To achieve this speed requirement, most security applications are developed on hardware platforms. BF-based security applications are no exception. To satisfy the high-speed demands of such applications, efficient hardware implementation of BFs is essential. Unfortunately, this issue has been neglected so far, and BFs and similar data structures are implemented as-is, without taking advantage of the underlying hardware.

In this paper, we address two issues regarding hardware BFs: (1) the query throughput and (2) memory fragmentation when multiple BFs are used for many small sets. We then use Aggregated Bloom Filters (ABFs) to address these two issues by leveraging the query mechanism so that the query throughput is optimized and the memory is reduced. The proposed scheme is verified with a simulation for NIDPS.

The rest of the paper is organized as follows. Section II describes the related work. Section III outlines the issues addressed in the paper. Section IV describes the proposed scheme. Section V analyzes the proposed scheme. Section VI discusses possible attacks on the ABF. Section VII provides performance results. Section VIII concludes the paper.

II. RELATED WORK

BFs are space-efficient data structures for membership queries on a set $S$ of arbitrary keys. A BF is a bitmap of $m$ bits and all of these bits are initially set to 0. $k$ different universal hash functions are used to map a key value to $k$ of the $m$ bitmap positions.

To insert a key into the BF, the key is first hashed using each of the $k$ hash functions to get $k$ bitmap positions. Then these bits at these positions are all set to 1. To query for a key ($i.e.$, test whether the key is a member of the set, $S$), again the key is hashed using each of the $k$ hash functions to get $k$ bitmap positions. If any of the bits at these positions is 0, the key is not in the set. If, on the other hand, all these bits are set to 1, then either the key is in the set, or the bits have been set to 1 during the insertion of different members of the set, causing a false positive (FP). An FP is the false identification of a key as a member of this set, although it is not. The FP $f$ of a BF for a set of $n$ items can be given as [9]

$$f = \left(1 - e^{-kn/m}\right)^k$$

Applications of BFs can be divided into two broad categories (1) Update-intensive and (2) Query-intensive. Update-intensive applications such as DDoS and network forensics update the bits corresponding to a key (or update bins for a Counting Bloom Filter (CBF) [9]) completely or partially and sometimes test the existence of a key in the BF. In query-intensive applications, such as NIDPS [2], [3] and virus scanning [4], the set programmed to the BF is either static or pseudo-static, which requires no or infrequent updates in the BF. Thus, determining whether the query key is a member of this set or not is the main operation for these applications. In this paper, we focus on query-intensive applications. Query operations are the most time-consuming operation for these applications and improving query performance improves the overall application performance. However, BFs are generally implemented as-is, without taking advantage of the underlying hardware.
In [10], Kocak and Kaya proposed to divide the hash functions for the BF into two pipeline stages, where the hash functions in the second stage are calculated only when the hash functions in the first stage all match. As a result, the overall power consumption of the BF is reduced. Our approach here is similar to their approach in the sense that we also propose to calculate the hash functions sequentially. Yet, our approach differs from theirs since our scheme also allows parallel queries to be aggregated and applied at the same time. Additionally, our aim is to improve the throughput of the system rather than decreasing power consumption, which is the main goal of [10].

In [11], the authors describe their hardware implementation of an NIDPS using BFs. Network intrusions can be identified by characteristic strings in the attack packets called signatures. Each intrusion has its own unique signature. In [11], the authors program all signatures with the same length to one BF, thus using one BF for each possible signature length. This causes fragmentation and under-utilization of on-chip memory. Although in [12], the authors combined signatures with different sizes into a single BF, their approach requires truncating all signatures in a group to the shortest signature length in the group. This causes a higher false-positive rate for longer signatures, since suffixes of these signatures are ignored.

III. PROBLEM DEFINITION: DESIGNING OPTIMAL HARDWARE BLOOM FILTERS

A. BF Query-Speed Optimization for FPGA Implementation

The performance of a BF for query-intensive applications can be solely determined by the performance of the query function. This is because for static sets, no updates are required and for pseudo-static sets, the updates are infrequent and their effect on performance is negligible. Thus optimizing query performance improves the throughput for most of these applications. For a query, the BF only gives a match if all bit values at addresses determined by all the hash functions are read and if they are all ones. For software applications, the query function for BFs are implemented as a series of hash functions. These hash functions are calculated on the input (i.e., query) one at a time and the hash result is used as an address to a bitmap. Then, the bit value at this address is read. As soon as one bit value is found to be zero, the query is immediately discarded, without calculating the remaining hash functions. This way, software implementations avoid any unnecessary hash function calculations or memory accesses. Thus, they are optimal in terms of the number of hash functions calculated per query. As a result, their speed is also optimal. We should note here that we stretch the definition of the term optimal a little bit. Actually, one can argue that by having prior knowledge about the distribution of queries and hash functions, one can predict which of the hash functions will result in a zero bit. Ideally then, discarding a query will take only one hash function operation, so optimally only one hash per non-matching queries is required. However, here we assume no such prior knowledge and assume for any query, all hash functions are equally likely to result in a zero bit. We use the term optimal to mean to discard non-members as soon as one hash function resulting a zero bit is calculated in an environment where the hash function calculation order is not a function of queries.

However, in hardware this is not always the case. For hardware implementations, BFs are implemented as parallel structures to improve throughput. In other words, when a query arrives, all hash functions are calculated, the corresponding bitmap locations are read concurrently, and the bit values read from the bitmap are ANDed. The result of this AND function determines if the BF gives a match (AND output is one) or a no match (AND output is zero). This way, due to hardware parallelism, a query can be calculated much faster. However, this does not mean that parallel BFs in hardware are optimal in terms of speed. Just one zero in the bitmap results is enough to declare a no match. Yet, in most cases, many bitmap locations have zeros, where all but one of these results are redundant. This is a waste of processing power. Eliminating these redundant calculations, will allow parallel hardware BFs to reach optimal speed. In this paper, our first goal is to develop a hardware architecture to eliminate this redundancy, thus to maximizing the throughput of hardware BFs.

B. BF Size Optimization for FPGA Implementation

For a BF given with a fixed $m/n$ ratio, the optimal number of hash functions, $k$ to achieve low false-positive probability can be derived as [9]

$$k = \lceil \ln(2) \times m/n \rceil$$

Note that, (2) suggests that for an arbitrary $n$ value to achieve an optimal BF (i.e., a BF with low false-positive probability using a small memory and a small number of hash functions), $m$ and $k$ need to be selected carefully. The largest embedded memory on FPGAs are Block RAMs (BRs) usually consisting of fixed-sized blocks with a constant number of read/write ports. Although some FPGAs [13] have embedded BRs with a few different configurations, they still do not allow arbitrary configurations. This pre-defined memory size and port configuration dictates a fixed $m/k$ ratio that is only optimal for a limited number of $n$ values. As a result, to have optimal BFs, the signature set size should satisfy certain constraints.

For instance, embedded memory in Xilinx Virtex 2E FPGAs is divided into 4096-bit fixed-size blocks, where each block is accessible through two dedicated ports. Thus, a BF using $B$ of these memory blocks has $k = 2 \cdot B$ hash functions and $m = 4096 \cdot B$ bits. These values are only optimal for a single set size, $n$ as

$$n = \ln(2) \times m/k = \ln(2) \times \frac{4096 \cdot B}{2 \cdot B} \approx 1419$$

If this BF is not used for this particular $n$ number of items, either the memory is underutilized or there is a higher false-positive rate compared to the optimal case. Additionally, in
most cases, many BFs are required for a typical application with a varying $n$. For instance, for NIDPS, ideally one BF for each signature length in the signature set is needed. Most of these per length subsets are very small with varying lengths causing memory fragmentation.

The number of available ports determines the query throughput on parallel BFs, if each port is dedicated to only a single BF. If this port is shared by two BFs, the throughput is automatically halved if concurrent access is needed to both BFs (which is generally the case). Thus, even if the memory size requirement for a given subset is very small, the whole BR is dedicated to a single BF to achieve maximum throughput. Dedicating even a single memory block on an FPGA for these sets is overkill and causes low memory utilization. Our second goal in this paper is to aggregate these subsets in a way that they can share the same memory with little or no degradation in throughput, thus improving memory utilization.

IV. AGGREGATED BLOOM FILTERS

In this section, we describe our proposed Aggregated Bloom Filter (ABF) and show how ABF improves query throughput and save memory over Standard Bloom Filters (SBFs).

A. Bloom Filters For Aggregated Queries

Queries are excerpts of packets coming from a high-speed link. Based on the application, queries can also be associated with packets or query locations in the packet. Parallel hardware implementations of SBFs process one query at a time. To remove the redundancy, i.e., by allowing at most one zero output for any query, we first serialize the queries so that hashes for each query are calculated sequentially. Then queries are aggregated (i.e., one BF processes multiple queries at the same time) to achieve better average throughput while worst-case throughput remains the same as the SBF. To support aggregated queries, and thus improve query throughput, we propose the hardware architecture given in Figure 1. In the core of this new architecture, there is a BF, that consists of $k$ hash functions ($H_1 \ldots H_k$) and $k$ corresponding bitmaps ($B_1 \ldots B_k$) each with $m_i = m/k$ bits memory. Let’s call each hash function-bitmap pair a processing element (PE). Each $PE_i$ serves a corresponding query queue ($QQ_i$). The Query Distributor is responsible for distributing queries to the QQs evenly. A counter ($C_i$) for each query ($Q_i$) is responsible for counting matches received for that query (not shown in the figure).

The programming operation here is similar to the programming of the SBF. The queries are processed as follows. When a query arrives at the ABF, the Query Distributor chooses the shortest (i.e., least occupied) $QQ$ and writes the new query to this queue. Each $PE_i$ starts with processing the head item in its $QQ_i$ to determine whether it has a match to this query. If so, the $i$-th match signal, $M_i$, is set to 1. If there is no match in $PE_i$ to a query $Q_A$, $Q_A$ is abandoned and $PE_{(i+1)}$ will process the next query available in $QQ_{(i+1)}$ in the next machine cycle. If there is a match in the $PE_i$ for $Q_A$ in the next machine cycle $PE_{(i+1)}$ will process $Q_A$ and the $C_{(i+1)}$ will hold the number of hash functions calculated so far for the $Q_A$ query. If all the hash functions are calculated and a match for $Q_A$ is returned (i.e., $C_i = k-1$ and $PE_i$ gives a match for $Q_A$), query $Q_A$ is a match. Note that all matching queries as well as their corresponding counter values are shifted to the next $PE$ at each machine cycle, whereas a matching query and its counter at the last $PE$ will be rotated to the first $PE$.

B. Set Aggregation to Increase Multi-Bloom Filter System Throughput

Some applications require multiple BFs for different data sets. For instance, in NIDPS, one BF is required for each possible signature length ($1 \ldots L_{max}$), where $L_{max}$ is the maximum signature size. This requires querying $L_{max}$ BFs, each with $n_i (i = 0 \ldots L_{max} - 1)$ different items and $m$ bits in parallel thus requiring $L_{max} \cdot k$ memory accesses. This causes most of these memories to be under-utilized. Here we propose to aggregate multiple sets (i.e., $M$ sets), as Figure 2 shows, into a single BF with size $m$ bits. If SBFs are used for this purpose with the same technology since number of memory ports available will also be a factor of $M$ less compared to the parallel case, the throughput will be degraded by a factor of $M$. On the other hand, if an ABF is used, aggregation reduces

1A machine cycle is defined as the time required for a PE to process a query.
the number of total queries to achieve similar throughput with $1/M$th of the original memory. Additionally, original worst-case throughput can be guaranteed by replicating $M$ ABFs above. This approach will still have a better average throughput and the same memory and worst-case throughput. This way, the ABF allows a memory-speed throughput that is not possible for parallel SBFs.

V. Analysis

Assume a BF with $m$ bits and $k$ hash functions is used to store a set $S$ with $n$ items. Denote the number of zeros in this bitmap as $z$ and the number of ones as $e$ so that $m = e + z$. Now suppose this BF is going to be used for membership queries on $S$ in the following way.

To determine whether a given query is a member of the set $S$, up to $k$ different hash functions ($h_1 \ldots h_k$) are calculated sequentially for this query. Each hash function selects one bit from this bitmap, $b_1 \ldots b_k$. The hash functions are not calculated in parallel. Instead, they are calculated sequentially, starting from the bit selected by $h_1$. If this selected bit has a value zero, the query is a non-member and immediately discarded (i.e., a true negative). Otherwise, the next hash function is calculated. If all hash functions select a bit with a value one, then the query is claimed to be a member of the set (i.e., a true positive). Although, it can also be a non-member misidentified as a member (i.e., false positive). A true- or false-positive query definitely requires $k$ hash operations. However, for a true-negative query, the decision can be made earlier (as discussed above). The expected number of hash functions required for a true-negative query can be calculated as follows.

The number of bits selected after $k$ hash operations with value zero follows the binomial distribution with parameters $k$ and $z/m$ and the expected number of bits with value zero after $k$ hash operations is

$$E[k,0] = \frac{zk}{m} \quad (4)$$

Similarly, the expected number of bits with value one after $k$ hash operations is

$$E[k,1] = \frac{ek}{m} \quad (5)$$

Then, let $\Phi \leq k$ defines the hash function index such that the first hash function with bit value zero is $h_\Phi$. Then the probability that $\Phi = \phi$ is

$$p_\phi(\Phi) = \begin{cases} \sum_{i=0}^{\phi-2} \left( \frac{E[k,1] - i}{k - i} \right) \cdot \frac{E[k,0]}{k - \phi + 1}, & \text{if } \phi \leq E[k,1] - 1 \\ 1, & \text{else} \end{cases} \quad (6)$$

As a result, the expected number of hash operations required to discard a non-member query is equal to the expected $\Phi$ and can be given as

$$E[\phi] = E[\Phi] = \sum_{i=0}^{k} i \times p_\phi(i) \quad (7)$$

For an optimal BF, $e \approx z \approx m/2$. For the optimal case, $E_k[0] = E_k[1] = k/2$. Then from (6),

$$p_{\Phi,\text{opt}}(\phi) = \begin{cases} \sum_{i=0}^{\phi-2} \left( \frac{k/2 - i}{k - i} \right) \cdot \frac{k/2}{k - \phi + 1}, & \text{if } \phi \leq k/2 - 1 \\ 1, & \text{else} \end{cases} \quad (8)$$

Given a query set with $q$ queries with $f$ false positives and $t$ true positives, the expected number of hash operations required to complete the queries on this set is

$$N_h(\text{ABF}) = k \cdot (t + f) + E[h] \cdot (q - (t + f)) \quad (9)$$

If instead an SBF is used with the same parameters, the number of hash functions required is

$$N_h(\text{SBF}) = k \times q \quad (10)$$

So, the average improvement in the number of hash operations required in the aggregated case with respect to SBF can be given as

$$\frac{N_h(\text{SBF})}{N_h(\text{ABF})} = \frac{q}{t + f} + \frac{k \cdot q}{E[h]} \left(q - (t + f)\right) \quad (11)$$

Since most of the normal traffic is expected to be clean from attacks unless there is an ongoing Denial-of-Service attack, $q \gg (t + f)$. Then (11) becomes,

$$\frac{N_h(\text{SBF})}{N_h(\text{ABF})} \approx \frac{k}{E[h]} \quad (12)$$

VI. Attacking ABF: Mimicking Worst-Case Query Profile

An attacker may learn the performance corresponding to different queries by sending traffic and measuring the response time of the NIDPS. This way, an attacker may come up with a query set that results in worst-case behavior.

Note that the starting hash function index for a query is random. As a result, the number of hash functions calculated for a key is different each time the key is queried. This reduces the predictability of the system. Yet, certain queries map to many ones and only a few zeros have, on average, a longer query time. An attacker may learn queries with longer query times. Similarly, an attacker may learn the false positives (which is also possible for SBFs), though they are harder to learn.

There are two possible solutions for this attack (1) Using a small cache to store recent queries with longer query times and (2) periodically, rehashing the BF so that the query performances as well as false positives are changed.
VII. PERFORMANCE

We developed a software model of ABF in C++ to verify its performance. Up to 16-byte signatures from a current Snort signature set were programmed into the ABFs. A packet trace from MIT Lincoln Labs [14] (first 10,000 packets from Monday, Week 1, The Attack Free Traffic Trace) was used to test the performance of the proposed scheme.

First, we tested the query performance of ABF using BF parameters \(k_i = 8\) and \(m_1 = 16,384\) as suggested in [3]. A set of 15 BFs corresponding to signature lengths 2 to 16 bytes are queried with the packet trace. Note that network traffic generally includes many sequences of zeros (e.g., for padding purposes). These zero sequences cause many matches in the BFs. Luckily, the zero sequences in the input can easily be detected using a simple circuit that counts the number of consecutive zeros at the beginning of the input window, \(z_c\). Once \(z_c\) is determined, any query with length \(l \leq z_c\) can bypass the BFs, since we know these queries are zero sequences with length 2 to \(z_c\). For this reason, these queries are ignored in our results for both SBF and ABF. Additionally, to achieve even query counts for every BF, the packet traces are assumed to be a single continuous stream. This way more queries are generated than normal, but each BF receives an equal number of queries, making the comparison among BFs easier. Figure 3 shows the number of total hash operations required for each BF when queried with the MIT packet trace. This graph shows a 7-fold decrease in the number of hash operations required for ABF with respect to the case where SBFs are used. Secondly, to show the instantaneous hash operation requirements of ABF, the number of total hash operations required in all ABFs for each byte of the input is shown in Figure 4. From this figure, it can be seen that an average ABF query requires 15.62 hash operations, and 95% of the ABF queries require at most 17 hash operations. No query in ABF requires more than 37 hash operations, whereas SBF always requires 120 hash operations per input byte.

The above measurements are given for the optimal BF parameters introduced in [3]. In Figures 5 and 6, the simulations are repeated for the same signature set and packet trace with different BF parameters for a fixed false-positive probability of \(10^{-5}\) and fixed \(n/m\), respectively. In these graphs, the hash operations required are normalized to compare for different \(k\) values on the same graph. Here, we show percentage of zero results corresponding to each query. This is a measure of the likelihood of early discard of queries (when a 0 value is hit). For both cases, 90% of the queries result in at least 60% zeros in the \(k\) hash operations.

![Fig. 4. Distribution of the total number of hash operations required for all the BFs for MIT packet traces.](image)

Next, we show the effectiveness of grouping different signature length sets into a single BF as proposed in Section IV-B. We arbitrarily grouped four consecutive-length signatures into one \(BF\). The item counts are shown in Figure 7 where aggregated sizes are close to the optimal value of 1419. Note that further grouping is also possible. Here, the total memory size is close to a quarter of the original memory size (since the last group does not have four subsets). Figures 8 and 9 show total hash counts and total hash functions required for each input byte, respectively.

VIII. CONCLUSION

In this paper, we propose a new Bloom Filter architecture, the Aggregated Bloom Filter (ABF), suitable for hardware implementation. We illustrated its performance in NIDPS. The
additional functionality used in this architecture uses simple logic. Our simulation results show seven-fold improvement in the average query throughput and four times less memory usage. The proposed methods can be applied to a wide range of applications that require hardware Bloom Filters. In this paper we only address the optimization of the query-intensive applications by using the ABFs with a single example application, NIDPS. We will investigate the performance improvement when using the ABFs in update-intensive applications in the future.

IX. ACKNOWLEDGMENT

We would like to thank to Lei Tian for his invaluable comments.

REFERENCES