Journal of Big Data

RESEARCH

Open Access

eBF: an enhanced Bloom Filter for intrusion detection in IoT

Fitsum Gebreegziabher Gebretsadik[1,2*], Sabuzima Nayak[2] and Ripon Patgiri[2]

*Correspondence:
fitsum_rs@cse.nits.ac.in

[1] School of Computing, Mekelle University, Mekelle, Tigray, Ethiopia
[2] Department of Computer Science and Engineering, National Institute of Technology, Silchar 788010, India

Abstract

Intrusion Detection is essential to identify malicious incidents and continuously alert many users of the Internet of Things (IoT). The constant monitoring of events generated from many devices connected to the IoT and the extensive analysis of every event based on predefined security policies consumes enormous resources. Accordingly, performance enhancement is a crucial concern of Intrusion Detection in IoT and other massive Big Data Applications to ensure a secure environment. Like many Big Data Applications, the Intrusion Detection system of the IoT needs to employ the fast membership filter, Bloom Filter, to quickly identify possible attacks. Bloom Filter is an admiringly fast and space-efficient data structure that quickly handles elements of extensive datasets in small memory space. However, the trade-off between the query performance, the number of hash functions, memory space, and false positive probability remains an issue of Bloom Filter. Thus, this article presents an enhanced Bloom Filter (eBF) that remarkably improves memory efficiency and introduces new techniques to accelerate the filtering of malicious URLs. We experimentally show the efficacy of eBF using a real Intrusion Detection dataset. The experimental result shows that the proposed filter is remarkably memory efficient, faster, and more accurate than the state-of-the-art filters. eBF requires 15.6x, 13x, and 8x less memory compared with Standard Bloom Filter, Cuckoo filter, and robustBF, respectively. Therefore, this new system significantly enhances the performance of Intrusion Detection of IoT that concurrently monitors several billion events crosschecking with the defined security policies.

Keywords: Bloom filter, Intrusion detection system, IoT, Big data

Introduction

The Internet of Things (IoT) refers to the massive network of integrated "Things" on the Internet. The "Things" refers to the physical objects that vary from simple household objects to sophisticated industrial tools embedded with mechanical and digital machines, computing devices, sensors, and people or animals provided with biochip [1, 2]. IoT is a combination of many synchronized technologies that efficiently support the day-to-day activities of human beings [3].

IoT interlinked 11.3 billion devices by 2021 and forecasts the number will increase to 29 billion by 2030 [4]. This massive interaction handles Big Data and supports every human activity. The rapid growth of IoT connectivity and ubiquitous technology

integration creates a conducive ecosystem for the applications of smart cities, logistics, transportation, health care, and home appliances. However, as the network interlinks heterogeneous sources with correlating events, the risk of exposure to intruders becomes high [5]. So, ensuring the security of IoT networks from attackers is a significant concern for IoT development [1, 2, 6].

Several Intrusion Detection systems (IDSs) have already been proposed to protect the IoT from attackers [7]. Figure 1 depicts the conventional model of IDS. The IDS monitors every event in the network and analyzes it as per the predefined possible indications of security policy threats or violations. Intrusion is prevented by determining the sign and stopping the detected incident. IDS plans to notify network users about incoming attacks by continuously monitoring the network traffic. Nevertheless, the vast amount of data accessed and analyzed according to defined signs needs advanced techniques that enhance processing efficiency. The fast and compact membership data structure- Bloom Filter [8]—supports an efficient Intrusion Detection process providing a true or false match based on hashed bits. Bloom Filter is well known for supporting several Big Data processing systems [9].

Bloom Filter is a space-efficient data structure implemented to boost the performance of searching an element in an extensive dataset using small memory space with high speed. Bloom Filter applies bit-wise data representation, consuming low memory space to handle a large number of queries of Big Data applications. Besides, Bloom Filter implements hash functions to generate a separate digest for every element representation efficiently and uniquely. So, data entry to and data extraction from the filter based on the hashing requires linear complexity of time O(1) [10].

Standard Bloom Filter supports insertion and lookup operations. The introduction of variants changes the classical features to allow deletion operation and speed up the processing capacity. Moreover, this progress reduces the memory space requirement to a relatively better size. Nevertheless, using small memory space for a Big Data key representation creates a false positive error. Because Bloom Filter is a probabilistic data structure, it can return a true or false result with a certain probability. The probability of returning a true result for a key not existing in the data set is known as the false positive probability (FPP). Thus, the advantage of using low memory representation in a bit representation causes a problem of determining a non-existent element as a member of the set. Besides, the trade-off between time and space requires

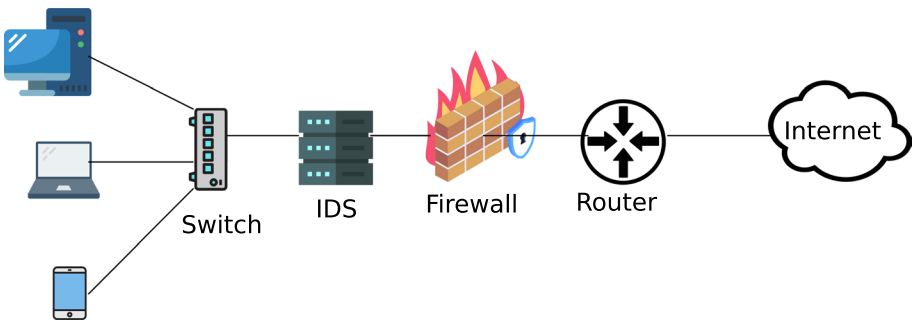


Fig. 1 Conventional intrusion detection system as a component of network security

careful remedy. Accordingly, several studies [10–14] proposed variants of Bloom Filter. Cuckoo filter [15] was introduced to replace the standard Bloom Filter by avoiding the efficiency and accuracy limitations. robustBF [14] is a space and time-efficient multidimensional Bloom Filter variant compared to the Standard Bloom Filter, Counting Bloom Filter, and cuckoo filter. robustBF is also more accurate than these filtering variants. So, the introduction of powerful variants makes Bloom Filter an essential performance optimization data structure for Big Data processing.

Contribution

The main goal of our proposed system is to enhance the current bloom filter variants to efficiently and effectively support Intrusion Detection in IoT. Our key objectives are as follows-

- To significantly reduce memory space consumption of the Intrusion Detection system.
- To enhance insertion and lookup speed by implementing efficient algorithms.
- To preserve the lowest rate of false positive occurrences without sacrificing the memory footprint and performance of the filter.

The Intrusion Detection system is a central traffic filtering system. However, the connected device also requires an Intrusion Detection system at their end, for instance, smartphones. Bloom Filter can fulfill such demands. Therefore, we propose a Bloom Filter-based model for an Intrusion Detection system that fits with IoT devices. This article presents eBF, an Intrusion Detection system that implements a new highly efficient Bloom Filter variant with a deep learning model. Training a deep learning model requires enormous computation resources such as GPU; however, testing is faster and does not require many computational resources as compared to the training process. A trained deep learning model can be deployed in IoT devices for Intrusion Detection. However, the deep learning models are heavyweight and it demands a reduction of the load on the deep learning model. Therefore, it requires additional data structures to reduce memory consumption and faster lookup performance. Consequently, we use a Bloom Filter with a deep learning model that can be deployed in IoT devices. The Bloom Filter can respond to the queries already learned and reduces the unnecessary loads on the deep learning model. Notably, we use the deep learning model as a black box and focus more on Bloom Filter for enhancement of the Intrusion Detection System. This new system introduces a new method to significantly improve the efficiency of previous Bloom Filter variants. In addition to controlled synthetic datasets, eBF uses real Intrusion Detection datasets from IoT to test its applicability in Intrusion Detection for IoT and other major Big Data applications.

We summarized the contribution of our proposed work by comparing it with the Standard Bloom Filter [8], Cuckoo Filter [15], and robustBF [14] as follows-

- eBF is a memory-efficient variant that uses only 15.6x, 13x, and 8x less memory space compared to Standard Bloom Filter, Cuckoo Filter, and robustBF, respectively.
- The insertion speed of eBF is the fastest of all the tested state-of-the-art membership filters. It reveals that eBF demands 5x, 1.25x, and 1.28X less time than the time used by the Standard Bloom Filter, Cuckoo Filter, and robustBF, respectively.
- eBF exhibits its sole advantage in avoiding unwanted searches by minimizing the searching time of disjoint data sets to 3.45X less than the Standard Bloom Filter time, 1.69X less than the Cuckoo Filter searching time, and1.31X less than the robustBF searching time.
- eBF speed advantage increases with the increase in the dataset sze. This confirms that eBF is relevant to Big Data processing.
- eBF yields almost zero False Positive Probability

Organisation

This article consists of corresponding sections that illustrate the importance and significance of the proposed system. Section 2 precisely demonstrates the fundamental features and basic operations of a Standard Bloom Filter. Section 3 discusses related previous studies and explains the relevance of the new system. Section 4 also demonstrates the model of the proposed method by presenting the algorithms. Besides, section 5 depicts the experimental outcome by presenting comparison graphs. Section 6 precisely discusses the distinctive features of the proposed Bloom Filter variant. Finally, section 7 winds up the presentation of this article with conclusive messages.

Bloom filter

Bloom Filter is a probabilistic data structure for efficient membership testing of an element from an extensive dataset. Bloom Filter implements bit-wise data representation to avoid fetching the big dataset from permanent storage to memory. So it uses minimal memory space to run millions of search operations on an extensive dataset. Besides, Bloom Filter applies fast and robust hash functions to generate representation hashes with the lowest collision probability.

Parameters of bloom filter definition

The main parameters on which Bloom Filter's definition and performance depend are memory size *(M)*, number of hash functions *(K)*, and the probability of false positive occurrences (FPP). The maximum limit of FPP can be predefined based on the system's behavior. The increase in the number of elements *(N)* that the Bloom Filter represents directly affects the memory size and the number of hash functions. Equations 1 and 2 deliver the optimal memory space and the number of hash functions required to develop a powerful Bloom Filter [14].

$$M = -\frac{N\ lnFPP}{(ln2)^2} \tag{1}$$

The increase in K diminishes the time efficiency of the Bloom Filter. The decrease in K also has the probability of increasing FPP. So the optimal number of hash functions is computed as:

$$K = \frac{M}{N} ln2 \tag{2}$$

If the system developer does not predefine the FPP, equation 3 computes the maximum false positive errors compromised to ensure the optimal time and memory efficiency of the Bloom Filter.

$$FPP = \left(1 - \left(1 - \frac{1}{M}\right)^{KN}\right)^{K} \tag{3}$$

Bloom filter operations

Standard Bloom Filter allows insertion and searching operations. Figure 2 visualizes the architectural model of a Standard Bloom Filter. The insertion function adds a representation of an element to the allocated memory space of the Bloom Filter, and the search function checks the existence of an element representation in the Bloom Filter.

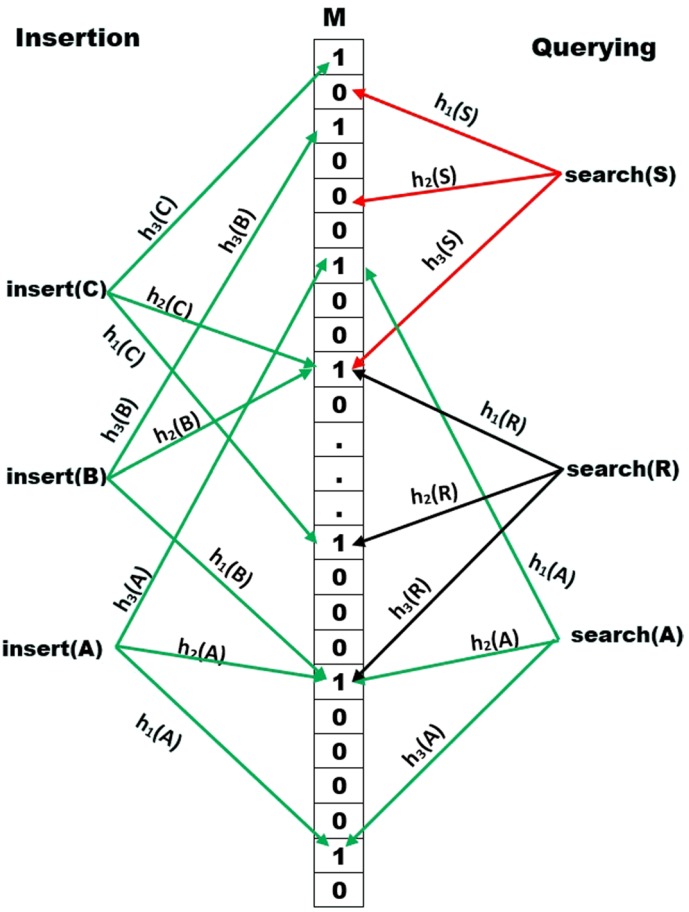


Fig. 2 Architecture and operations of Standard Bloom Filter

Introducing several variants of the Bloom Filter with the intention of efficiency and accuracy enhancement adds various features to the standard Bloom Filter. However, the basic features and operations remain core factors in the enhanced Bloom Filter variants to implement an appropriate computational optimization algorithm for several Big Data Applications.

Next to memory space allocation for the Bloom Filter, every memory unit is initialized to zero. To insert new elements, Bloom Filter uses one or more hash functions. Based on the hash functions' result, the 0 value of the corresponding cells of the filter is changed to 1. Figure 2 shows that three hash functions (h_1, h_2, *and* h_3) are used to insert *A*, *B* and *C* into the Bloom Filter. Similarly, searching employs these three functions to check the existence of *A*, *R*, and *S*. So, both insertion and searching operations of a standard Bloom Filter use the same hash functions. In the case of searching the Bloom Filter, the element is hashed by the hash functions. The corresponding cells of the filter are checked. If all cell values are 1, then the element exits.

Correct results of membership filter

Big Data System employs Bloom Filter to check the existence of an element in the storage. So, the expected output of the Bloom Filter algorithm is either True (confirming the existence of the requested element) or False (denying the existence of the element). Accordingly, from Fig. 2 we can observe that *search(A)* returns *true* because *A* was inserted by *insert(A)*. Moreover, the figure depicts that *search(S)* returns *false* denying the membership of *S*. This result is a true answer because the cells expected to be hashed by h_1 (S) and h_2*(S)* both hold *0 s*, showing that *S* is not a member of the dataset inserted in the Bloom Filter.

Probabilistic data structure

Probabilistic data structures compromise uncertain answers to lessen the trade-offs between space and time. Probabilistic behavior is relevant for developing customized algorithms per the system's need. Some access control systems may allow visible false positives for account creation to avoid similarities of accounts. Unlikely, customer service that can serve hundreds of millions of clients may strictly diminish false positives near zero to prevent intruders and discard unwanted processes. As a result, using probabilistic data structure has a vital role in enhancing computational efficiency.

Bloom Filter applies probabilistic components to efficiently process the data it holds but cannot provide a definite answer (exactly true/false). Though it is possible to reduce the degree of uncertainty to near insignificant impact, the "true" result may not always mean the element is a member of the specified set. Standard Bloom Filter does not have an issue with false negatives. Nevertheless, Unlike the standard Bloom Filter, some variants, such as the Counting Bloom Filter, face a probability of denying the availability of an existing element giving out a false negative [16]. Hence, several research works deliver plenty of solution schemes to enhance the accuracy of the algorithms.

False positive

The main challenge of a Bloom Filter on which many researchers have been working is false positive [17, 18]. When the number of elements *N* stored in the bit array with

a fixed size M increases, the probability of representing two or more elements using a single bit increases due to hashing collision. Consequently, membership of all the elements hashed to the same location M_i will always reply true for the existence of the not inserted element. That is why the name of this problem is known as false positive. In Fig. 2, *search(R)* returns *true* confirming the presence of R. However, the result is incorrect because all 1's hashed in the cells corresponding to the hash functions of element R ($h_1(R)$, $h_2(R)$, and $h_2(R)$) map representation of elements from A, B and C not representation of element R.

Consequently, in addition to enhancing time and memory efficiency, minimizing *FPP* is the main issue of studies related to Bloom Filters. The formula to find optimal FPP based on N number of elements inserted to an M size array of Bloom Filter based on K different hash functions is eq. 3.

Hash function

Hash function digests a data input to its compressed size to significantly enhance the processing performance. Cryptographic techniques implement hashing algorithms to secure data. However, non-cryptographic hash functions, including JenkinsHash and MurmurHash, are more time-efficient than cryptographic hash functions [10, 19]. Accordingly, Bloom Filter algorithms prefer implementing non-cryptographic hash functions because cryptographic hash functions reduce the processing speed and do not reduce the false positives [20]. For instance, the Bloom Filter in Fig. 2 uses three hash functions (h_1, h_2, *and* h_3) to insert and to lookup an element to and from the Bloom Filter respectively. Hashing helps the Bloom Filter to handle Big Data of complex systems concisely.

Performance and accuracy trade-off

Bloom Filter must always deal with how to overcome the competitive challenge between efficiency and accuracy. The accuracy of a Bloom Filter degrades when FPP increases. Increasing the number of hash functions to insert an element is a remedy for reducing FPP. However, increasing the number of hash functions increases computational overload and consumes more memory space, negatively affecting the critical advantage of using a Bloom Filter in small memory space. Hence, a better system that can balance this trade-off without significant negligence on performance and accuracy is the dominant research topic on Bloom Filter. Hence, this article introduces an extraordinarily efficient and accurate Bloom Filter that can enhance the processing performance of Intrusion Detection in IoT and other Big Data Applications.

Related works

Intrusion Detection is a highly active research area with several implications. Studies show that Intrusion Detection has a significant economic impact on the computerized industries [5]. The employment of the classical data structures is not sufficient to efficiently and effectively handle the extremely increasing size of data in cyberspace [21]. Consequently, various advanced techniques and algorithms have evolved for decades. Probabilistic data structures are among the advanced data structures that highly support the applications that manage Big Data processing [22]. Probabilistic data structures

are essential approaches to resolving the delay that occurs due to the extremely large size of data processing in systems like cloud computing, financial systems, and social media that simultaneously engage billions of users' interactions. Probabilistic data structures use non-encryption hash functions, including murmur hash, to facilitate the access of elements in a lower memory space [10]. So the advantage of using probabilistic data structures to handle the searching process is more critical in terms of time and space efficiency when compared with traditional data structures. However, the issue of delivering inaccurate results remains a trade-off with the time and space advantages. So, the probability of error occurrences needs minimization to an insignificant level. Big Data applications widely employ Bloom Filter and its variants as basic time and space efficient probabilistic data structure techniques [23, 24].

A Bloom Filter in Intrusion Detection uses to determine if a given event's data (e.g., network packet) is a member of the predefined set of threats stored in the database. A standard Bloom Filter uses bit arrays to store the representations of the set of threats. The hash result of the predefined threat is used as an index of the bit array to set the value of the memory unit to 1. As a result, Intrusion Detection systems implement the Bloom Filter to efficiently and effectively identify possible security attacks. Hence, using an enhanced Bloom Filter is very important to protect the billions of devices linked to the IoT from malicious attacks [25, 26].

Artan et al. [27] proposed a variant of Bloom Filter known as Aggregate Bloom Filter to support network Intrusion Detection systems efficiently.

Groza and Murvay [26] implemented Bloom Filer on an Intrusion Detection system to identify potential attacks in the controller area network that monitors and reports a large number of traffic attacks. The result shows that Bloom Filter is a vital tool for effectively handling Intrusion Detection processes within a constrained resource. Besides, Bala et al. [28] demonstrated the significance of using Bloom Filter to efficiently handle the massive amount of spam observed in SMTP sessions. This work used an Intrusion Detection system to detect spamming bots of SMTP sessions related to the social network users of a university campus.

The Intrusion Detection systems of IoT and other complex Big Data applications use Bloom Filter for its suitable approach of representing sets and supporting efficient searching execution. Zinkus et al. [29] designed an Intrusion Detection system that employs Bloom Filter to efficiently handle fuzzy anomaly detection in IoT. The evaluation result of the system shows that the detection of simulated attacks is well enhanced. Hence, developing an enhanced Bloom Filter that supports an Intrusion Detection system of IoT is necessary. Lucchesi et al. [30] employed Bloom Filter to design an optimized IP lookup algorithm. Because of Bloom Filter, the proposed algorithm highly optimizes the throughputs of the IP lookup. The above studies justify that Bloom Filter is an important data structure to handle Big Data and efficiently support the Intrusion Detection of networks. However, the proposed schemes implemented the Standard Bloom Filter, which requires further enhancements.

Standard Bloom Filter is helpful in approximation representation. However, similar to other probabilistic data structures, it faces false positive issues. Thus, several studies proposed improved schemes to:

1 Reduce the memory space required to represent relatively huge data economically.
2 Increase the performance of search processing by implementing high-speed algorithms.
3 Lessen the false positive rate to an insignificant degree or near zero.

Counting Bloom Filter [31] is a Bloom Filter variant that implements a counter bit in addition to the representation bit. Unlike the standard Bloom Filter, counting Bloom Filter allows deletion operation. Every corresponding counter increments or decrements when an element is inserted into or deleted from the filter. Harwayne-Gidansky et al. [25] presented an Intrusion Detection system based on a Counting Bloom Filter (FPGA SoC) to achieve a scalable and high degree of throughput. Despite its advantage of including deletion operation for some applications that require removal of elements, counting Bloom Filter introduces memory overhead and consumes more processing time [12, 22]. In addition, the Counting Bloom Filter shows a high degree of FPP that degrades the system's accuracy.

CountBF [12] enhances the time and memory efficiency of standard Bloom Filter and counting Bloom Filter, holding a low FPP. A *r*-multidimensional Bloom Filter (rDBF) [13] is proposed, which has a significantly fast filtering algorithm with lower memory space and fewer false positives. This scheme introduced a new view of hashing that requires the X and Y coordinates to minimize the trade-off between memory space and FPP. Unlike the Standard Bloom Filter, this multidimensional variant avoids the dependency on the number of hash functions. The decrease in the number of hash functions without degrading the accuracy quality boosts the processing speed.

The introduction of Cuckoo Filter [15] targeted to replace the use of the Standard Bloom Filter. Cuckoo Filter highly enhanced the capacity of the traditional Bloom Filter. Accordingly, Mosharraf et al. [9] used Cuckoo Filter to enhance the searching performance of distributed Big Data Applications. The proposed scheme doubled the performance of the search in the targeted Big Data clusters. Cuckoo Filter has time and memory efficiency advantages over the Standard Bloom Filter [8]. However, elements can get rid of the insertion queue and be placed in an alternative bucket as a result that increases insertion time. The rapid growth in size and complexity of Big Data Applications requires a continuous engagement in performance improvising methods.

robustBF [14] is the 2-dimensional feature of rDBF [13]. robustBF implements the modified murmur hash function to enhance processing speed, ensure high accuracy, and diminish the FPP near zero. The scheme consumes lower memory space than the Standard Bloom Filter and Counting Bloom Filter. Nevertheless, memory consumption needs more enhancement for better efficiency. Besides, the insertion and search speed requires improvement to cope with the rapid growth of the IoT Domain. The Intrusion Detection systems used in the IoT also require an enhanced Bloom Filter that can accelerate the threat determination to serve the fast-growing number of devices.

To finalize, the exponential growth of data size and complexity of IoT and similar Big Data Applications demand continuous enhancement of supporting algorithms. Hence, this article demonstrates an enhanced Bloom Filter that integrates extremely

efficient programming techniques that significantly reduce the memory space used by the state-of-the-art. Besides, this new variant introduces an efficient way of implementing algorithms to minimize processing time and maintain the lowest FPP without staining performance.

Proposed system

We propose a novel intrusion detection technique for IoT devices, called eBF, which implements a Bloom Filter. eBF is based on a two-dimensional Bloom Filter that filters many malicious activities without sacrificing memory footprint. It uses a tiny memory footprint making it suitable to integrate into IoT devices. Furthermore, it requires fewer computing resources for its operations.

Figure 3 depicts our proposed architecture. eBF uses two Bloom Filters, namely, *iBF* and *bBF*. The *iBF* stores the information of intrusion data packets, and *bBF* stores benign data packet information. The key embodiment of our proposed work is both *iBF* and *bBF* cannot contain the same data packet. If both Bloom Filters contain the same data packet then it guarantees that it is a case of false positive. Therefore, we query the incoming data packet in *iBF* and *bBF* for false positive cases. We have four scenarios: (a) maybe benign: *iBF* return false and *bBF* returns true, (b) maybe

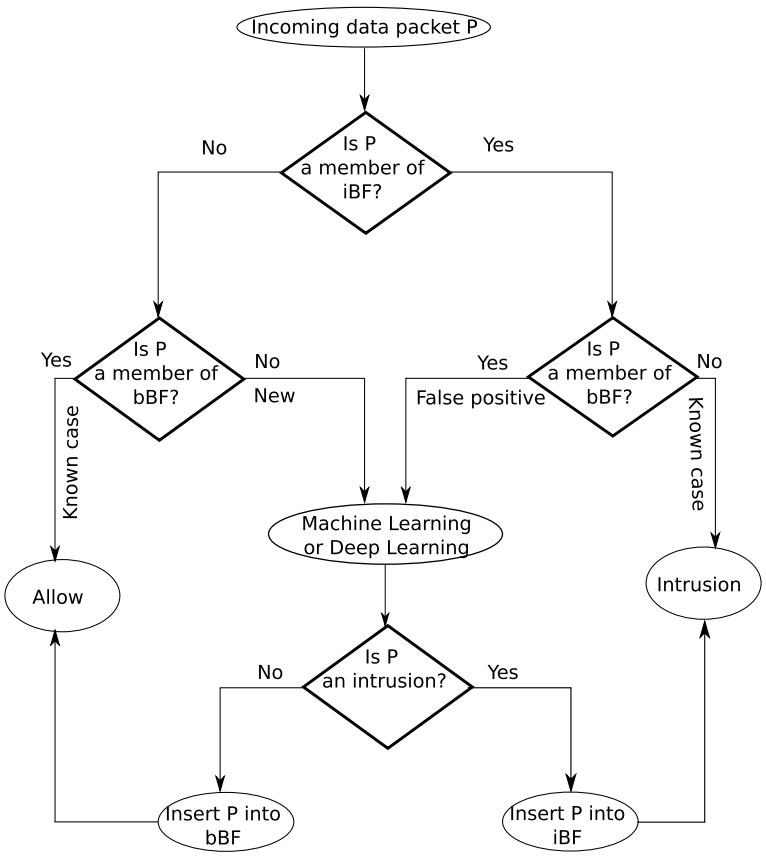


Fig. 3 Flow-chart of our proposed system for IDS

intrusion: *iBF* return true and *bBF* returns false, (c) new data packet: *iBF* return false and *bBF* returns false, and (d) false positive: *iBF* return true and *bBF* returns true. For (c) and (d) scenarios, the data packet requires the intervention of a deep learning model.

An incoming data packet is queried for membership in *iBF*. If *iBF* returns false for a data packet, then it queries to *bBF*. If *bBF* returns true for the same data packet, then it is a benign data packet. The system can proceed with the data packet for further processing. If the data packet is a member of *iBF* but not a member of *bBF* then it is an intrusion. Therefore, the data packet is blocked from further processing. If a data packet is not a member of both the Bloom Filters, i.e., *iBF* and *bBF*, then it is a new data packet. Therefore, the data packet is forwarded to the deep learning model for classification. Based on the classification of the deep learning model, the data packet is inserted into either *iBF* or *bBF*. If the deep learning model classifies the new data packet as intrusion then it is inserted into *iBF* for future references and also, is blocked from further processing. Otherwise, the new data packet is inserted into *bBF* and allowed for further processing. Since the Bloom Filter can have a false positive; therefore, both *iBF* and *bBF* can return true for the same data packet. Hence, the data packet is forwarded to the deep learning model for correct classification.

The key focus of this work is to design an efficient Bloom Filter that provides a faster query response time using a small memory footprint without sacrificing its performance in eBF. The Bloom Filter is an exceedingly memory-efficient two-dimensional Bloom Filter compared with the state-of-the-art Bloom Filter. It is an enhanced version of robustBF [14]. Figure 4 illustrates the architecture of the two-dimensional Bloom Filter. robustBF allocates 64-bit per cell of the two-dimensional integer array whereas its enhanced version employs 32-bit per cell. Hence, the total number of cells increases in our proposed Bloom Filter even if the memory size is reduced. The memory size is eight times smaller than robustBF. Therefore, our proposed Bloom Filter performs better than robustBF in reduced memory footprint. Moreover, our proposed Bloom Filter is relatively faster and at least equally accurate as robustBF. Similar to the robustBF, we use the dimensions of the filter to speed up the insertion and query process without degrading its performance.

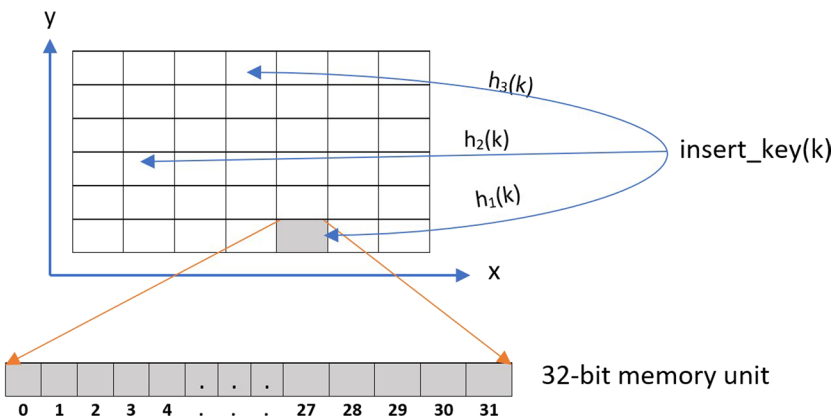


Fig. 4 Two-dimensional memory architecture of eBF. The dimensions are *X* and *Y* where each cell of the two-dimensional array is 32-bit

Our proposed Bloom Filter uses three hash functions for its operations. In the case of a query, the number of hashing depends on the response of the Bloom Filter. For instance, if the first-bit position is set to zero, then it does not check the next-bit position and it concludes that the key is not a member of the Bloom Filter. Otherwise, it checks the rest bit positions. Therefore, the number of the hash function can vary from one to three in a query operation.

Operations

Algorithm 1 initializes all parameters of the Bloom Filter *(BF)*. This algorithm accepts the expected number of keys *(N)* to insert with the required *FPP*. Then it determines the size of memory space *(M)* by calculating the maximum *(X)* and *(Y)* dimensions of the two-dimensional memory structure.

Algorithm 1 Initialization of Bloom Filter

```
1: procedure INIT(N, FPP)
2:     Determine X and Y dimensions
3:     Allocate X × Y memory space
4:     Initialize BF cell to zero
5:     return BF
6: end procedure
```

Algorithm 2 demonstrates the insertion operation. Let, $BF_{i,j}$ be the particular cell in the Bloom Filter which is a 32-bit integer value initialized with zero. Let k be the input key to insert into the Bloom Filter. We calculate $i = \text{Murmur2}\,k, length(k), seed \bmod X$ and $j = \text{Murmur2}k, length(k), seed \bmod Y$ which gives us the precise location of the cell. Again, the bit position in the cell $BF_{i,j}$ is calculated as $d = \text{Murmur2}k, length(k), seed \bmod 31$. The d^{th} position of $BF_{i,j}$ is set to one. Then repeats the same procedure for the next two bit positions by changing the seed value.

Algorithm 2 Insertion of key representation

```
1: procedure INSERT-KEY(BF, k, seed)
2:     for i : 1 to 3 do
3:         h_v = MURMUR2(k, length(k), seed)
4:         i ← h_v mod X
5:         j ← h_v mod Y
6:         d ← h_v mod 31
7:         BF[i][j] ← BF[i][j]|(1U << d)        ▷ Inserting key representation
8:         seed = h_v
9:     end for
10: end procedure
```

Algorithm 3 checks the representation of the key in all three corresponding memory units using three Test(). The algorithm requires key k. It uses murmur2 hash functions to generate cell locations similar to the insertion operation. The Test() returns true if the bit location calculated by the hash function is 1; otherwise false. In this algorithm, if the first Test() returns false, then Lookup() returns false and does not execute the other two Test(). The second Test() is executed if the first Test() returns true. Similarly, if the second Test() returns false Lookup() returns false; otherwise, execute the third Test(). Hence, each Lookup function has a varying number of hash functions which reduces the

query time. Moreover, the Lookup function is more efficient in the case of an absent key rather than a present key.

Algorithm 3 Lookup for membership of a key

```
1: procedure LOOK-UP(BF, k)
2:     h_1 ← MURMUR2(k, length(k), seed)
3:     if TEST(BF, h_1) = true then
4:         seed = h_1
5:         h_2 ← MURMUR2(k, length(k), seed)
6:         if TEST(BF, h_2) = true then
7:             seed = h_2
8:             h_3 ← MURMUR2(k, length(k), seed)
9:             if TEST(BF, h_3) = true then
10:                return True
11:            else
12:                return False
13:            end if
14:        else
15:            return False
16:        end if
17:    else
18:        return False
19:    end if
20: end procedure
```

Experimental result

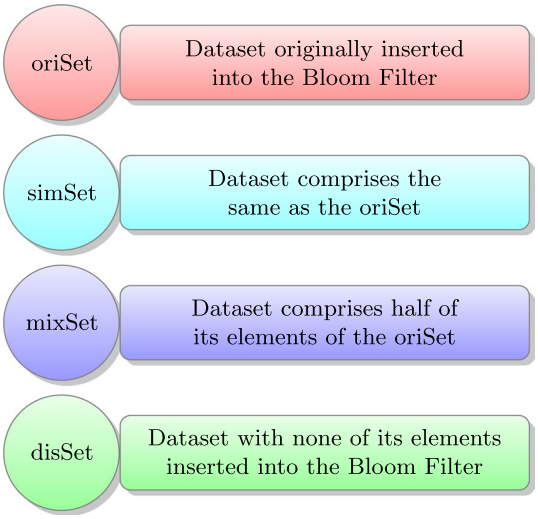
This section demonstrates the performance of eBF in comparison with other state-of-the-art Bloom Filter variants. The efficiency and accuracy of eBF are compared to Standard Bloom Filter, Cuckoo Filter, and robustBF. The system used to test eBF consists of a processor with the specification of Intel®Core™ i5-8250U CPU @ 1.60GHz × 8, a memory of size 8GB, and a 1TB hard disk. The operating system is a 64-bit Ubuntu 22.04 LTS.

Dataset description

The experiments are conducted using both real data saved in CSV file format and synthetic data to evaluate the accurate rate of false positive occurrence across the different filtering systems. Though we have considered the FPP tested in the uncontrolled real dataset for selecting the number of hash functions, the accuracy assessment is accurate using the synthetic data. The reason to use a synthetic dataset for accuracy assessment is that the data is known to conclude on the similarity or unlikeness.

Synthetic datasets

The synthetic datasets that we use in the evaluation are collections of integers generated in a way that they can use for the accurate evaluation of the system performance. The experiment uses three types of datasets to test the accuracy and efficiency of the proposed filter. An original dataset, i.e., *oriSet*, for instance, O={o_1, o_2, o_3, ..., o_n} is input to the Bloom Filter. Then, keys of other datasets, i.e., *simSet*, *mixSet*, and *disSet* is queried to eBF to verify their existence.



The first testing dataset is a set of keys similar to those represented in eBF. So, it is known as **simSet**, for example, S=$\{o_1, o_2, o_3, ..., o_n\}$. The second testing dataset is known as **mixSet**, for example, M=$\{o_1, o_2, o_3 ... d_1, d_2, ..., d_n\}$. Its half content is intentionally changed to differ from the original dataset inserted in the Bloom Filter. The **disSet** is the third dataset which consists of completely different keys from the original set (*oriSet*), for example, D=$\{d_1, d_2, d_3, ..., d_n\}$.

Real datasets

The real datasets used to assess the performance of the proposed scheme are available in a public repository. These real datasets are related to IoT-detected intrusions from different systems at different times. An experiment that uses real datasets increases the feasibility of the proposed system on the IoT and similar Big Data Applications. Table 1 shows a brief description of the real datasets used in this experiment.

The datasets DSet1 (Downloaded from [32]) and DSet2 (Downloaded from [33]) contain network traffic sniffed from nine IoT devices using Wireshark in a local network using a central switch. It includes two Botnet attacks: Mirai and Gafgyt. The datasets contain 23 statistically engineered features extracted from the.pcap files. Seven statistical measures (variance, mean, magnitude, count, covariance, radius, and correlation coefficient) are considered in the experiments. The dataset DSet3 (Downloaded from [34]) contains data generated from more than ten types of IoT devices, i.e., ultrasonic sensors,

Table 1 Details of real datasets

Dataset	Dataset description and download link	Size	Number of records
DSet1	Refined IoT dataset for Intrusion Detection systems Without duplication. [32]	143 MB	907996
DSet2	A selected dataset for evaluating deep learning-based Intrusion Detection systems. [33]	1.13 GB	2219202
DSet3	IoT dataset for Intrusion Detection systems. [34]	1.56 GB	7062607

low-cost digital sensors for sensing temperature and humidity, water level detection sensors, etc. The data is related to attacks of connectivity protocol and categorized into five threats: injection attacks, DoS/DDoS attacks, man-in-the-middle attacks, information gathering, and malware attacks [33].

The experiment uses a single and more relevant column of every dataset as a key of representation. Hence, the keys of the datasets are represented by the unique key columns, i.e., "ID" of DSet1, "Triggering time" of DSet2, and "variance" from DSet3.

Hash function selection

The number and type of hash function significantly affect a Bloom Filter's efficiency. Several hash functions are available, but murmur hash [35] is an efficient and effective non-cryptographic hash function [10, 14]. So, eBF employs the murmur hash function to achieve the goals of ensuring remarkably high time and space efficiency as well as optimal accuracy. An increase in the number of hash functions improves the accuracy by minimizing false positives; however, it reduces the insertion and lookup efficiency. On the other hand, speed increases by reducing the number of hash functions but increases the FPP. So, the determination of the number of hash functions demands appropriate evaluation. Accordingly, an experiment was conducted on the proposed system with a fixed memory space but various values of K using the synthetic dataset, the result is showcased in Table 2.

A single hash function, i.e., $K = 1$ has the fastest speed for inserting keys. However, $K = 1$ generates some false positives. Implementing two hash functions $K = 2$ scores the second fastest time but records zero false positives. Thus, implementing $K = 2$ provides optimal accuracy with acceptable time efficiency using a synthetic dataset. Nevertheless, synthetic data alone cannot lead to a conclusion.

The efficiency and accuracy evaluation of a different number of hash functions on the real dataset is important in deciding the number of hash functions. So, using $K = 2$ on real data has $FPP = 0.002$. But using $K = 3$ has better results which exhibits $FPP = 0.0006$. Therefore, we decided to implement $K = 3$ to achieve the best performance with at most $FPP = 0.001$.

Table 2 Comparison of insertion speed (seconds) and false positive probability (FPP) based on the number of hash functions in eBF using synthetic dataset.

Number of *K*	10 Million		50 Million		100 Million	
	Speed	FPP	Speed	FPP	Speed	FPP
1	1	0	5.7	0.00016	10.5	0.0001
2	1.35	0	6.27	0	13.5	0
3	1.7	0	8.65	0	17.35	0
4	2.15	0	11	0	22	0
5	2.37	0	12	0	23	0
6	2.48	0	13	0	25.8	0
7	2.8	0	14.6	0	29.3	0
8	3.1	0	16.3	0	32.9	0
9	3.5	0	18	0	36.6	0
10	3.9	0	19.9	0	40.7	0

10, 50, and 100 Million is the number of elements inserted into eBF

Experiments using synthetic datasets

This section demonstrates the experiments conducted on eBF to compare with Standard Bloom Filter, Cuckoo Filter, and robustBF using synthetic datasets. These controlled datasets effectively evaluate the accuracy of the Bloom Filter variant. This is because the similarity or disparity of two or more datasets can be adequately identified only when the dataset elements are known. Therefore, the synthetic datasets that we use to test the membership filters contain 10 Million (10 M), 50 Million (50 M), and 100 Million (100 M) keys. Evaluating the performance of insertion and lookup operation on millions of keys makes shows our proposed approach is efficient and fast to handle Big Data processing.

Memory space

The low memory footprint is one of the basic factors that make the Bloom Filter preferable to apply in Big Data Applications. However, it is important to enhance the capacity of the Bloom Filter to diminish the memory space required to represent huge data. Figure 5 highlights the comparison of eBF, Standard Bloom Filter, Cuckoo Filter, and robustBF based on memory footprint. The eBF consumes 15.6X, 13X, and 8X less memory compared with the Standard Bloom Filter, Cuckoo Filter, and robustBF, respectively. Figure 5 depicts the advantage of using eBF over the state-of-art to improve the memory space efficiency of Intrusion Detection systems in IoT and query processing in Big Data Applications.

Insertion time

Fig. 6 depicts the comparison among eBF, Standard Bloom Filter, Cuckoo Filter, and robustBF based on insertion time. In all datasets, Standard Bloom Filter took the highest insertion time whereas eBF took the least time. The eBF took 5×, 1.25×, and 1.28× less compared to the insertion time of the Standard Bloom Filter, Cuckoo Filter, and robustBF for 100 million keys. Similarly, eBF takes an average of 1.28 times less time compared to the insertion time of robustBF.

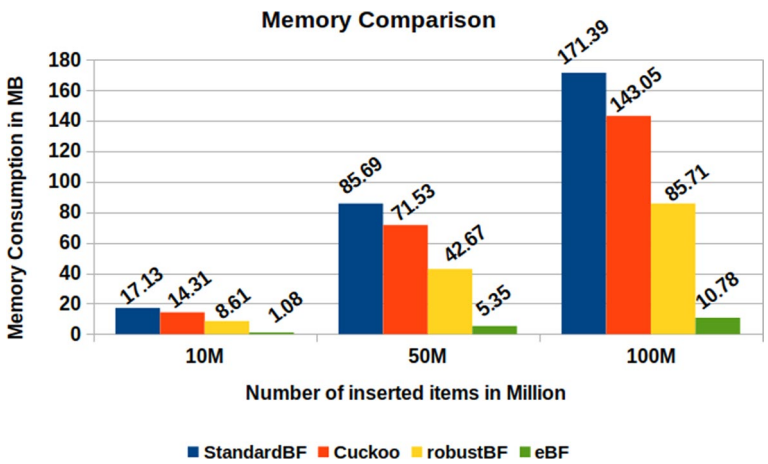


Fig. 5 Memory space comparison among the state-of-the-art membership filters using a synthetic dataset of 10 million, 50 million, and 100 million records. The lowest memory size consumption is the best

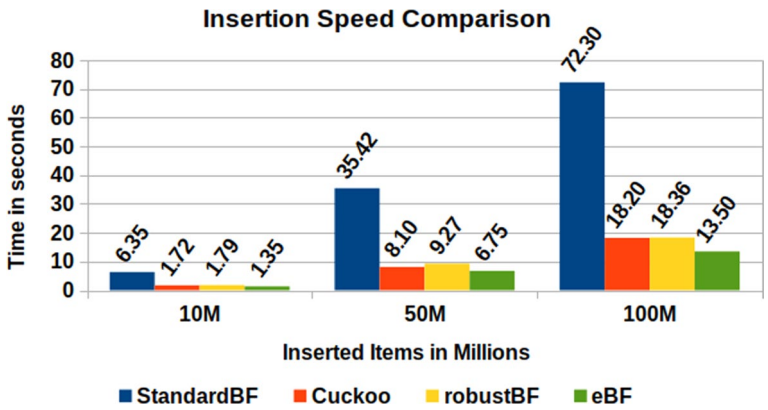


Fig. 6 Insertion time comparison among the-state of-the-art membership filters synthetic dataset of 10 Million, 50 Million, and 100 Million records. The lowest is the best

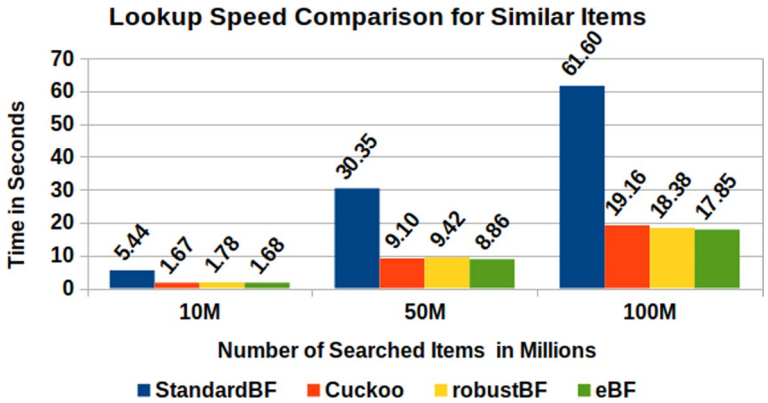


Fig. 7 Lookup speed comparison for *simSet* among Standard Boom Filter, Cuckoo Filter, robustBF, and eBF. Lower is better

Lookup time

Lookup speed is the most critical time factor because it is repetitively performed to check the existence of keys in the filter. For instance, if an event invokes access in the IoT, it must be crosschecked against the predefined threats. So, for every event generated from all devices of the IoT system, there are millions of searches to identify its reliability. These generated synthetic datasets are relevant to accurately distinguish the speed difference in lookup and precisely show the FPP in every scheme under the comparison process. Figure 7 displays the result of the speed comparison of the lookup of *simSet* in the Bloom Filter. The lookup time of eBF is 3.45×, 1.03× ×, and 1.03× less compared to Standard Bloom Filter, Cuckoo Filter, and robustBF for 100 million keys.

The query speed for datasets that are different from the original dataset is faster than the lookup of *simSet*. This difference comes from an algorithm we designed to ignore the process of the succeeding hash functions when the preceding hash function returns false. The negative return from one hash function assures the absence of the element. As a result, the eBF avoids unwanted searching. For instance, taking the dataset with 100 M and the second fastest system robustBF as comparing parameters, eBF uses 1.03X less than the time used by robustBF while lookup *simSet*. However, eBF only requires 1.11X

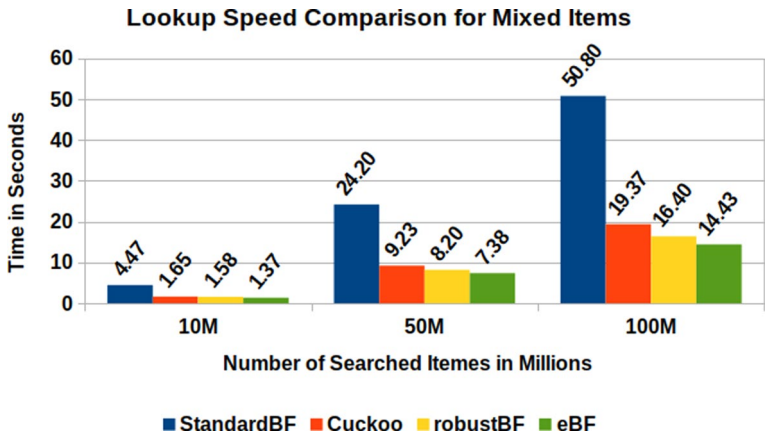


Fig. 8 A search speed comparison among filters on a dataset that proportionally consists of both similar and different elements compared with the elements of the original dataset represented in the Bloom Filter. Lower is better

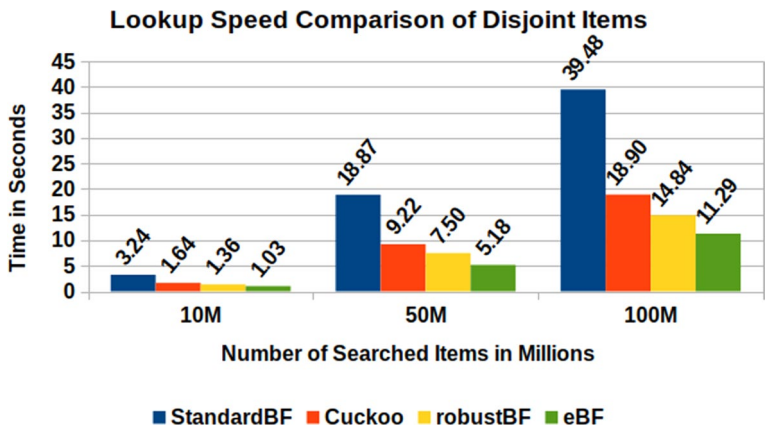


Fig. 9 Lookup time comparison among the filters using the disjoint dataset (*disSet*). Lower is better

and 1.31X less than the time used by robustBF for lookup *mixSet* and *disSet*, respectively. Accordingly, Fig. 8 displays the graphical illustration of lookup speed comparison using *mixSet*.

Besides, Fig. 9 also shows how the speed of disjoint dataset (*disSet*) lookup in eBF is the fastest of all queries of the same dataset in Standard Bloom Filter, Cuckoo Filter, and robustBF. Hence, eBF is not only extremely memory efficient but also more time-efficient when compared with the state-of-art membership filters.

Accuracy assessment

Accuracy assessment evaluates the correctness of the system to deliver a valid answer to users when lookup for the existence of an element in the system. Though the Bloom Filter is an efficient data structure for membership lookup, it faces the challenge of providing false positives. Accordingly, measuring the rate of false positives is the main aim of this experiment.

Based on the result of the experiment, the proposed system is highly accurate. Our experiment shows that both eBF and robustBF record zero false positives. So it is possible to conclude that the result approves the 100% accuracy of both eBF and robustBF under synthetic datasets. However, Standard Bloom Filter and Cuckoo Filter show small false positives. The accuracy is calculated in terms of the ratio of the sum of True Positive (*TP*) and True Negative (*TN*) to the expected true result and the sum of false positive occurrence *FP* and False Negative *FN* from the result of the system's output [36]. All the systems tested in this paper record zero FN. According to eq. 4, Fig. 10 witnesses the result of accuracy assessment based on the test of the three different synthetic datasets.

$$Accuracy = \frac{TP + TN}{(TP + TN + FP + FN)} \tag{4}$$

Hence, it is possible to conclude that eBF is an approvingly efficient and accurate Bloom Filter to handle Big Data membership queries.

Experimentation using real dataset

In addition to the synthetic datasets, this experiment uses real datasets accessed from an open repository. As described in Table 1, these datasets contain Intrusion Detection results of IoT systems. So, this performance test is appropriate to decide the significance of our proposed variant to defend IoT systems from attacks.

Memory space comparison

Space efficiency is one of the significant features that make Bloom Filter among the essential performance enhancement tools of Big Data Applications. However, this new proposed Bloom Filter variant eminently diminishes the memory size required to store the representation of tens and hundreds of millions of elements. Figure 11 depicts that eBF consumes the smallest memory space when compared with evaluated schemes. Thus, eBF is an appropriate solution for enhancing the efficiency of Big Data processing for membership identification. Membership classification is also the main task of Intrusion Detection in IoT. Hence, eBF is appropriate to ensure security in IoT.

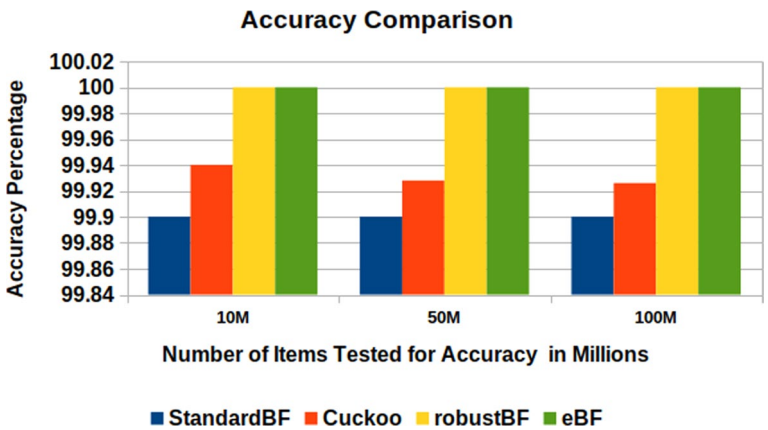


Fig. 10 Accuracy comparison between eBF and the state-of-the-art filters. Higher is better

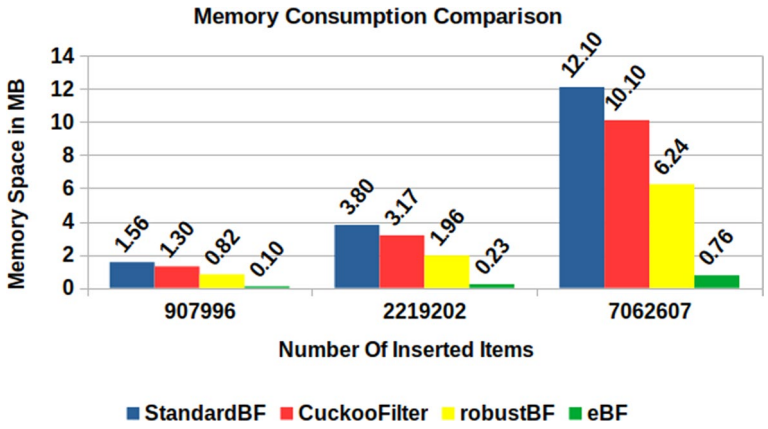


Fig. 11 Memory consumption comparison between eBF and the state-of-the-art membership filters. Real datasets are used in this comparison evaluation. Lower is better

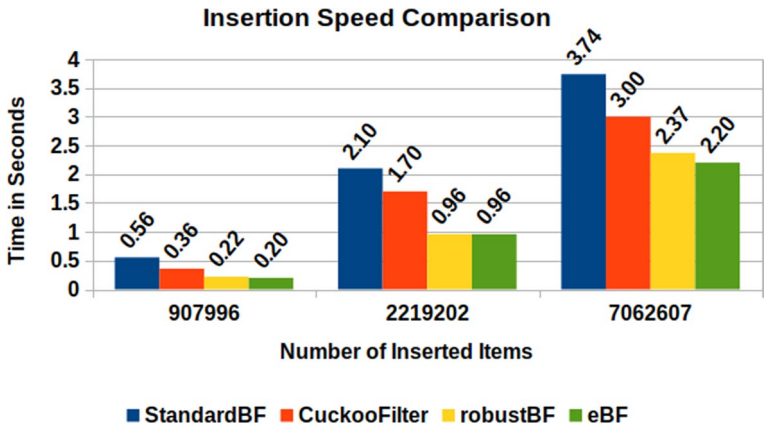


Fig. 12 Real data insertion speed comparison of eBF against Standard Bloom Filter, Cuckoo Filter, and robustBF. Lower is better

Speed comparison

The proposed variant is more time efficient than Standard Bloom Filter, Cuckoo filter, and robustBF. Figure 12 shows how the proposed system is faster than the other systems. The efficiency advantage increases when the number of elements represented in the filter increases. This quality makes eBF an appropriate tool for colossal data processing.

Figure 13 demonstrates the performance of query operations of the different membership filters for the real datasets. Though fast lookup is the distinguishing nature of a Standard Bloom Filter, Fig. 13 shows that lookup in eBF is more efficient than the state-of-the-art membership filter. Accordingly, eBF has become an efficient solution to optimize the performance of Big Data Applications.

Accuracy assessment

As shown in Table 1 the real datasets used for time and space assessment are accessed from different environments. However, they can contain similar elements. So, assessing

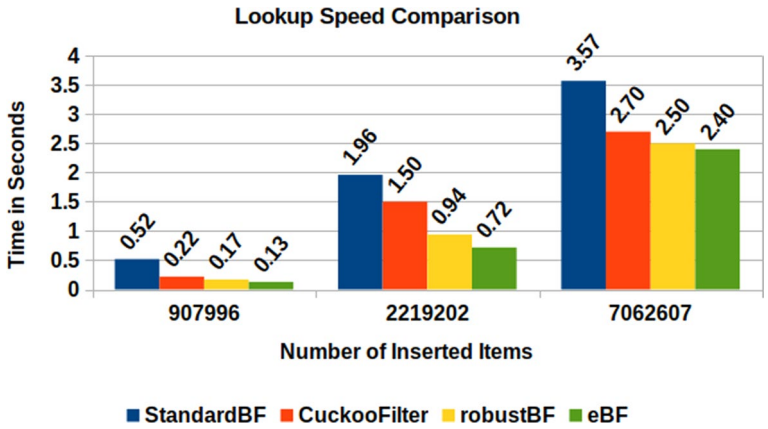


Fig. 13 Real data lookup speed comparison among eBF, Standard Bloom Filter, Cuckoo Filter, and robustBF. Lower is better

Table 3 Improvement of eBF over Standard Bloom Filter, Cuckoo Filter, and robustBF for 100 M dataset

Comparison with	eBF memory and time efficiency advantage				
	Memory efficiency	Insertion speed	Similar searching	Mixed searching	Disjoint searching
StandardBF	15.6×	5×	3.45×	3.33×	3.45×
CuckooFilter	13×	1.25×	1.03×	1.25×	1.69×
robustBF	8×	1.28×	1.03×	1.11×	1.31×

false positives is not as accurate as using synthetic datasets. It is possible to test the false negative status by querying the same dataset over the Bloom Filter that represents it. Accordingly, the experiment result reveals that Standard Bloom Filter, Cuckoo Filter, robustBF, and eBF recorded zero false negatives.

Comparison summary

The time and space comparison summary between the latest membership filters and our proposed variant - eBF is shown in Table 3. The table depicts that eBF outperforms all other membership filters in every aspect. For example, eBF uses 8× smaller than the memory used by the latest variant robustBF. The speed enhancement also shows that implementing eBF is better than the three evaluated methods. So, our proposed Bloom Filter variant is more efficient than the state-of-the-art membership filters to work with identifying huge data elements collected at every edge of the IoT system. This ability of eBF allows others to develop various powerful Intrusion Detection systems for IoT.

Discussion

IoT and various Big Data Applications handle an immense interaction among billions of users simultaneously. The Intrusion Detection Systems in IoT monitor malicious events based on a predefined set of threats to prevent billions of devices from being attacked. This complex interaction demands advanced technologies that enhance

computational performance [9]. Accordingly, eBF is the right solution to support the robust computation of IoT Intrusion Detection. Moreover, it can enhance the performance of complex networks of social media such as Facebook, YouTube, and WhatsApp and cloud vendors such as Microsoft, Amazon, and Google that handle the processing of Big Data. eBF surprisingly uses a tiny memory footprint and it is 15.6×, 13×, and 8× lower than the Standard Bloom Filter, Cuckoo Filter, and robustBF, respectively. Key insertion to and search processing on traditional Bloom Filter requires enhancement to cope with the rapid growth of applications. eBF is faster than the Standard Bloom Filter, Cuckoo Filter, and robustBF. eBF is a better solution to support the performance enhancement of Big Data Applications based on the insertion and search speed. Moreover, eBF shows better accuracy than all the systems tested on the same data and environment. Thus, this new system significantly addresses the efficiency and accuracy issues of Intrusion Detection in IoT.

Conclusion

In this article, we present a novel method to detect Intrusion Detection Systems for IoT, called eBF. Our proposed model relies on a deep learning model which is considered a black box. The key embodiment of our proposed scheme is to reduce the load of the trained deep learning model using a Bloom Filter because IoT devices cannot tolerate resource hunger computation.

We have carried out an extensive experiment to validate the performance of our proposed work with state-of-the-art filters. This significantly notable result was tested by using big datasets that amount to 100 million records. The datasets include actual data packets filtered from various IoT Intrusion Detection Systems. Besides, system-generated synthetic datasets with a set of integers were evaluated in a controlled way to expose the strength/weaknesses of the membership filters. The result shows that eBF is incredibly memory efficient using 15.6×,13×, and 8× less memory than the Standard Bloom Filter, Cuckoo Filter, and robustBF, respectively. eBF is also faster in inserting and searching operations than other membership filters. The experimental result exhibits that eBF is on average 5×,1.25×, and 1.28× faster than Standard Bloom Filter, Cuckoo Filter, and robustBF, respectively during insertion. The speed advantage of this new system increases more when there is searching for disjoint datasets. Accordingly, eBF is 3.45×, 1.69×, and 1.31× faster than Standard Bloom Filter, Cuckoo Filter, and robustBF, respectively. Hence, frequently requested unwanted searches can be avoided easily without affecting the system's performance. It also records zero false positives. This result shows that eBF is almost 100% accurate Bloom Filter. This proposed Bloom Filter variant has successfully achieved all the objectives of this article by delivering an enhanced Bloom Filter with notably exceptional performance and reliability for Intrusion Detection systems of IoT.

Author contributions
All authors equally contributed.

Funding
There is no funding information to disclose.

Availibility of data materials
The datasets used to test the performance of this system are available at 1 https://www.kaggle.com/azalhowaide/iot-dataset-for-intrusion-detection-systems-ids?select=BoTNeTIoT-L01-v2.csv. 2 https://www.kaggle.com/azalhowaide/iot-dataset-for-intrusion-detection-systems-ids?select=BotNeTIoT-L01_label_NoDuplicates.csv. 3 https://www.kaggle.com/datasets/mohamedamineferrag/edgeiiotset-cyber-security-dataset-of-iot-iiot

Declarations

Ethics approval and consent to participate
Not applicable

Competing interests
The authors declare that they have no competing interests.

Received: 30 August 2022 Accepted: 28 May 2023
Published online: 17 June 2023

References

1. Tewari A, Gupta BB. Security, privacy and trust of different layers in internet-of-things (iots) framework. Futur Gener Computer Syst. 2020;108:909–20. https://doi.org/10.1016/j.future.2018.04.027.
2. Yadav K, Gupta BB, Hsu CH, Chui KT. Unsupervised federated learning based ioT intrusion detection. In: 2021 IEEE 10th Global Conference on consumer electronics (GCCE). 2021;298–301 . 10.1109/GCCE53005.2021.9621784
3. Adel A. Utilizing technologies of fog computing in educational ioT systems: privacy, security, and agility perspective. J Big Data. 2020;7(1):1–29. https://doi.org/10.1186/s40537-020-00372-z.
4. Vailshery LS. Number of iot connected devices worldwide 2019-2030. Accessed Jul 2022.
5. Zuech R, Khoshgoftaar TM, Wald R. Intrusion detection and big heterogeneous data: a survey. J Big Data. 2015;2(1):1–41. https://doi.org/10.1186/s40537-015-0013-4.
6. Honar Pajooh H, Rashid MA, Alam F, Demidenko S. IoT big data provenance scheme using blockchain on hadoop ecosystem. J Big Data. 2021;8(1):1–26. https://doi.org/10.1186/s40537-021-00505-y.
7. Putra GD, Dedeoglu V, Kanhere SS, Jurdak R. Poster abstract: towards scalable and trustworthy decentralized collaborative intrusion detection system for ioT. In: 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI). 2020;256–257 . 10.1109/IoTDI49375.2020.00035
8. Bloom BH. Space/time trade-offs in hash coding with allowable errors. Commun ACM. 1970;13(7):422–6. https://doi.org/10.1145/362686.362692.
9. Mosharraf SIM, Adnan MA. Improving lookup and query execution performance in distributed big data systems using cuckoo filter. J Big Data. 2022;9(1):1–30. https://doi.org/10.1186/s40537-022-00563-w.
10. Patgiri R, Nayak S, Muppalaneni NB. Is bloom filter a bad choice for security and privacy? In. Int Conf Inform Network (ICOIN). 2021;2021:648–53. https://doi.org/10.1109/ICOIN50884.2021.9333950.
11. Patgiri R, Nayak S, Borgohain SK. Role of bloom filter in big data research: a survey. arXiv Preprint. 2019. https://doi.org/10.14569/IJACSA.2018.091193.
12. Nayak S, Patgiri R. Countbf: a general-purpose high accuracy and space efficient counting bloom filter. In: 2021 17th International Conference on Network and Service Management (CNSM). 2021;355–359 . 10.23919/CNSM52442.2021.9615556
13. Patgiri R, Nayak S, Borgohain SK. rdbf: A r-dimensional bloom filter for massive scale membership query. J Network Computer Appl. 2019;136:100–13.
14. Nayak S, Patgiri R. Robustbf: a high accuracy and memory efficient 2d bloom filter. arXiv Preprint. 2021. https://doi.org/10.48550/arXiv.2106.04365.
15. Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD. Cuckoo filter: practically better than bloom. In: Poceedings of the 10th ACM international on conference on emerging networking experiments and technologies. CoNEXT Ã¢â‚¬â„¢14, pp. 75Ã¢â‚¬â€œ88. Association for Computing Machinery: New York, NY, USA (2014). Doi:https://doi.org/10.1145/2674005.2674994
16. Guo D, Liu Y, Li X, Yang P. False negative problem of counting bloom filter. IEEE Trans Knowledge Data Eng. 2010;22(5):651–64. https://doi.org/10.1109/TKDE.2009.209.
17. Patgiri R. Hfil: a high accuracy bloom filter. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 2169–2174 (2019). https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00300. IEEE
18. Kiss SZ, Hosszu E, Tapolcai J, Ronyai L, Rottenstreich O. Bloom filter with a false positive free zone. IEEE Trans Network Serv Manag. 2021;18(2):2334–49. https://doi.org/10.1109/TNSM.2021.3059075.
19. Gerbet T, Kumar A, Lauradoux C. The power of evil choices in bloom filters. In: 2015 45th Annual IEEE/IFIP International Conference on dependable systems and networks. 2015;101–112. 10.1109/DSN.2015.21.
20. Patgiri R, Nayak S, Muppalaneni NB. Is bloom filter a bad choice for security and privacy? In: 2021 International Conference on Information Networking (ICOIN). 2021;648–653. 10.1109/ICOIN50884.2021.9333950.
21. Todorov Marinov M. A bloom filter application for processing big datasets through mapreduce framework. Int Confer Inform Technol (InfoTech). 2021. https://doi.org/10.1109/InfoTech52438.2021.9548638.
22. Singh A, Garg S, Kaur R, Batra S, Kumar N, Zomaya AY. Probabilistic data structures for big data analytics: a comprehensive review. Knowl Based Syst. 2020;188: 104987. https://doi.org/10.1016/j.knosys.2019.104987.

23. Kiss SZ, Hosszu E, Tapolcai J, Ronyai L, Rottenstreich O. Bloom filter with a false positive free zone. IEEE Trans Network Serv Manag. 2021;18(2):2334–49.
24. Harshan J, Vithalkar A, Jhunjhunwala N, Kabra M, Manav P, Hu Y-C. Bloom filter based low-latency provenance embedding schemes in wireless networks. IEEE Wireless Commun Networking Confer (WCNC). 2020. https://doi.org/10.1109/WCNC45663.2020.9120640.
25. Harwayne-Gidansky J, Stefan D, Dalal I. Fpga-based soc for real-time network intrusion detection using counting bloom filters. IEEE Southeastcon. 2009;2009:452–8. https://doi.org/10.1109/SECON.2009.5174096.
26. Groza B, Murvay P-S. Efficient intrusion detection with bloom filtering in controller area networks. IEEE Trans Inform Foren Secur. 2019;14(4):1037–51. https://doi.org/10.1109/TIFS.2018.2869351.
27. Artan NS, Sinkar K, Patel J, Chao HJ. Aggregated bloom filters for intrusion detection and prevention hardware. In: IEEE GLOBECOM 2007 - IEEE Global Telecommunications Conference. 2007;349–354 (2007). 10.1109/GLOCOM.2007.72.
28. Bala PM, Usharani S, Aswin M. Ids based fake content detection on social network using bloom filtering. In: 2020 International Conference on System, Computation, Automation and Networking (ICSCAN). 2020. 10.1109/ICSCAN49426.2020.9262360
29. Zinkus M, Khosmood F, DeBruhl B. Pidiot: probabilistic intrusion detection for the internet-of-things. IEEE Global Commun Confer (GLOBECOM). 2019. https://doi.org/10.1109/GLOBECOM38437.2019.9013264.
30. Lucchesi A, Drummond AC, Teodoro G. High-performance ip lookup using intel xeon phi: a bloom filters based approach. J Internet Serv Appl. 2018;9(1):1–18.
31. Fan L, Cao P, Almeida J, Broder AZ. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking. 2000;8(3):281–93. https://doi.org/10.1109/90.851975.
32. Kaggle's Non Duplicated IoT Dataset for Intrusion Detection Systems (IDS). https://www.kaggle.com/azalhowaide/iot-dataset-for-intrusion-detection-systems-ids?select=BotNeTIoT-L01_label_NoDuplicates.csv
33. Edge-IIoTset Cyber Security Dataset. https://www.kaggle.com/datasets/mohamedamineferrag/edgeiiotset-cyber-security-dataset-of-iot-iiot
34. Kaggle's IoT Dataset for Intrusion Detection Systems (IDS) With Duplication. https://www.kaggle.com/azalhowaide/iot-dataset-for-intrusion-detection-systems-ids?select=BoTNeTIoT-L01-v2.csv
35. Austin A. Murmurhash. Accessed Jun 2022.
36. Tharwat A. Classification assessment methods. Appl Comput Inform. 2020. https://doi.org/10.1016/j.aci.2018.08.003.

Publisher's Note