An Analysis of the Hardware-Friendliness of AMQ Data Structures for Network Security

Arish Sateesan[1(✉)], Jo Vliegen[1], and Nele Mentens[1,2]

[1] imec-COSIC/ES&S, ESAT, KU Leuven, Leuven, Belgium
{arish.sateesan,jo.vliegen,nele.mentens}@kuleuven.be
[2] LIACS, Leiden University, Leiden, The Netherlands

Abstract. Field-programmable gate arrays (FPGA) are increasingly used in network security applications for high-throughput measurement solutions and attack detection systems. One class of algorithms that are heavily used for these purposes, are approximate membership query (AMQ) data structures, which provide a mechanism to check, with a certain false positive rate, if an element is present in the data structure or not. AMQ data structures are used, for example, in distributed denial-of-service (DDoS) attack detection. They are typically designed to work efficiently on general-purpose processors, but when the high throughput of FPGAs is required, hardware-friendly implementations of AMQ modules are indispensable. A hardware-unfriendly AMQ module would considerably slow down the overall system and compromise the security when it is required to operate at line rate in a high-bandwidth network. Hence, choosing a suitable data structure and hardware architecture is of utmost importance. In this work, we propose FPGA architectures for various well-known AMQ data structures and analyze their hardware implementation properties. This work serves as a guideline on FPGA-based AMQ architectures for researchers and practitioners working on high-throughput network security applications on FPGA.

Keywords: Approximate membership query · FPGA · Network security

1 Introduction

As of 2022, every day approximately 5.6 billion Google searches are made, 90 million photos are shared on Instagram, 720,000 h of videos are uploaded to YouTube, 500 million tweets are posted, and a total of 2.5 Quintilian bytes of data are generated [1]. Increasing data rates induce hefty storage and computation costs. This issue becomes even more important in networking applications where data need to be processed at line rate, i.e., the rate at which data are transmitted in the network. The advent of Terabit Ethernet, i.e. Ethernet with speeds above 100 Gigabits per second (Gbps), leads to stringent throughput constraints, which are difficult to meet on resource-limited platforms.

L. Batina et al. (Eds.): SPACE 2022, LNCS 13783, pp. 287–313, 2022.
https://doi.org/10.1007/978-3-031-22829-2_16

Approximate membership query (AMQ) data structures are used for lookups in networking and database applications with strict speed requirements, memory limitations and/or power constraints [34]. As opposed to exact lookup architectures, AMQ algorithms have a small false positive rate. In a typical application, as shown in Fig. 1, AMQ data structures provide an estimate of whether an element (s), also called a key, is present in the data structure (S) not. AMQ data structures can be employed standalone when it is only necessary to know the presence of the key s. When a value needs to be read out as well, AMQ solutions are used in key-value storage mechanisms.

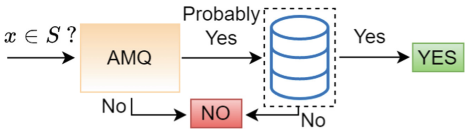


Fig. 1. Role of an AMQ data structure for lookup applications

In distributed denial-of-service (DDoS) attack detection systems, the lookup (AMQ) module checks if the parsed flow identifier or flow ID (f) of the incoming network flow is present in the blacklist or not. This is shown in Fig. 2. The flow is dropped if f is present in the blacklist. If f is not present, a detection module, based on, e.g., pattern matching, probabilistic algorithms or machine learning, examines the flow. Here, false positives of the AMQ lookup structure causes legitimate flows to be labelled as malicious. Therefore, the false positive rate (FPR) of the data structure becomes as important as speed. An example of a DDoS detection system that relies on a setup as shown in Fig. 2, is proposed by Scherrer et al. [33].

Membership queries or dictionary lookups have always been a challenging problem. This is even more so when the number of unique elements to be stored, commonly referred to as the *cardinality*, is very large. AMQ structures perform very well when the data set is static and the cardinality is known upfront. In this paper, our focus is on single and static data sets where the cardinality is already known or predictable. Data structures based on this criterion are the prime requirement for applications such as large flow detection mechanisms, which are used for DDoS detection, where measuring the network flow is stipulated to a specific measurement epoch. In network applications, the flow ID is the key for the lookup and the size of the flow ID is fixed. An IPv4 flow ID is characterised by the 5-tuple ⟨source address, destination address, source port, destination port, protocol⟩ and can be any combination of the 5-tuple. In this paper, the flow ID is taken as a combination of source and destination IP addresses and ports. We ignore the protocol field to keep the size of the flow ID to 96 bits. This means that the key input to the lookup or key-value storage mechanism is 96 bits in the scenario that we consider.

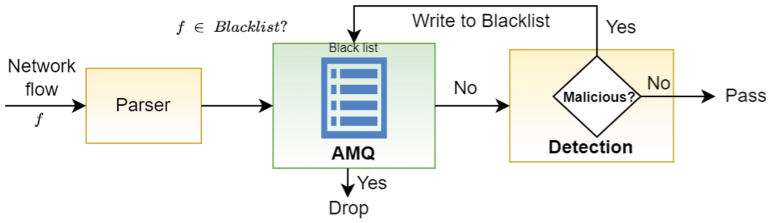


Fig. 2. Role of AMQ data structures for lookups in network attack detection systems

1.1 Challenges in Membership Query Data Structures on Hardware

On software platforms, a dictionary is the easiest solution for storing data as key-value pairs. However, a data structure similar to dictionaries would be very inefficient on hardware. The best available solution without any loss in accuracy for lookup data structures are Content Addressable Memories (CAM), which are too expensive for resource-constrained devices [31]. The operating frequency of a CAM decreases with increasing size, which is a problem when large amounts of data need to be processed at line rate. A large improvement over exact data structures, like CAMs, is offered by probabilistic lookup and key-value storage data structures, i.e., those that can have a small false positive rate in exchange for reduced lookup delay, resources and power consumption. Examples of probabilistic data structures are hash tables (in which false positives can be eliminated through chaining mechanisms), Cuckoo hash tables, and other hash-based techniques. Most AMQ data structures are improved versions of Cuckoo filters, which are derived from Cuckoo hash tables, or Bloom filters.

Even though probabilistic architectures come with many advantages such as lower memory requirements and lower lookup latency, the accuracy and hardware-friendliness are an important concern. When it comes to processing data at line rate on hardware, a number of challenges have to be addressed in order to maximize the accuracy and the lookup speed, and minimize the memory utilization. Data structures such as hash tables employed with linear probing or chaining as collision resistant mechanisms, would be difficult to implement on hardware as the size of the table would keep changing and the feasibility of pipelining is almost naught. Dynamic insertion is another requirement for network security applications, where a full set of flow IDs are not available at the time of construction and new flow IDs need to be added at runtime. Hence, static architectures are not preferred, unless all the required keys or rules which are to be stored are already available such as in regular expression matching.

In this work, we analyze existing probabilistic architectures to find a suitable probabilistic alternative for CAMs on hardware for network security applications. Our approach is to evaluate how suitable the algorithms are to be transformed into a hardware architecture, with a specific focus on achieving a speed-up on FPGA. The efficiency in software of these algorithms does not necessarily give us an idea of which algorithms perform the best in hardware. There are a plethora of such data structures available and analyzing each and every data structure

would be too much for this paper. Nevertheless, most of the well-performing data structures are derived from Bloom filters, Cuckoo filters, or hash tables, so we focus on the efficiency of these basic data structures in hardware. Our goal is to evaluate the hardware friendliness in terms of lookup latency, operating frequency, resource consumption, and suitability for pipelining.

2 An Insight into AMQ Data Structures

Starting from hash tables, numerous AMQ data structures have been proposed. The best known AMQ data structures are hash tables, Bloom filters [4] and Cuckoo filters [10]. Most of the prominent AMQ structures are either derived or optimized versions of these structures. Note that hash tables can be turned into exact lookup mechanisms when linear probing or chaining is applied. Besides the classification into Bloom filter based, Cuckoo filter based and hash table based data structures, there can be other ways of classifying, such as fingerprint-based/non-fingerprint based or static/dynamic architectures. Fingerprint-based data structures store either a short digest of the key or the entire key itself whereas non-fingerprint based data structures do not store the key/digest. Static architectures require the whole set of keys to be available at the time of construction, where dynamic architectures can support insertion and deletion of items on-the-fly. The term 'dynamic' may also be used to represent the data structures where resizing of the table at run-time is possible. However, in this paper we do not consider the dynamic resizing of the table at run-time. This is because of the fact that a complete reconstruction or rehashing of the data structure is required for dynamic resizing. This we try to avoid on hardware because it is purely an overhead and does not allow for a fair comparison.

Since we categorize all data structures into three types: hash table based, Bloom filter based and Cuckoo filter based, the remainder of this section elaborates on these three types. Table 1 shows a number of the prominent features of the basic data structures, which are relevant in network security applications. The table gives an average of the features from all three types. It is noted that there might be exceptions to these generalizations.

Table 1. Features of basic AMQ data structures

Data structure	Stores key	Stores fingerprint	Supports deletion	Supports lookups	Supports key-value store	Unlimited insertions
Hash table	✓	✗	✓	✓	✓	✗
Bloom filter	✗	✗	✗	✓	✗	✓
Cuckoo filter	✗	✓	✓	✓	✗	✗

2.1 Hash Table and Its Variants

Hash tables are the simplest and most conventional way to implement a lookup or key-value storage architecture. To add an item x to a hash table with m

locations, a hash function $h(x)$ is used to map the item to the table, where the key, value, or key-value pair are to be stored. A simple representation of the hash table is shown in Fig. 3(a). Before we dive deep into the details, we should clarify the difference between a hash table and a hash map. In a hash table the key is mapped to a location in the table using a hash function and the key is stored in that location. Hash map follows the same process but stores a key-value pair instead of a key. In this paper we use the term 'hash table' invariably for representing both hash table and hash map, and we refer to hash tables without collision resistance mechanisms in the form of linear probing or chaining.

Compared to exact associative array architectures such as CAMs, a hash table requires less memory for storing the same set of elements. However, hash tables are prone to collisions. Occurrences of hash collisions cause two different keys to be mapped to the same location in the table which causes data loss. The load factor α of hash table $\frac{n}{m}$ must be kept to a low value to reduce the collisions, where n and m are the total number of items to be added and the number of buckets in the hash table respectively. A bucket is a hash-indexed location in the table which could store one or more entries. The probability of at least a single collision in a hash table is $\frac{m!}{(m-n)!m^n}$ and the total average number of collisions is $\approx \frac{n^2}{2m}$. This means that there will be an average of $\approx$50% collisions if $n = m$. Hence, α must be lower than 0.5 to keep the collisions to a minimum, which would eventually cause under-utilization of memory. The memory efficiency of a hash table can be configured when the number of elements to be stored is known upfront.

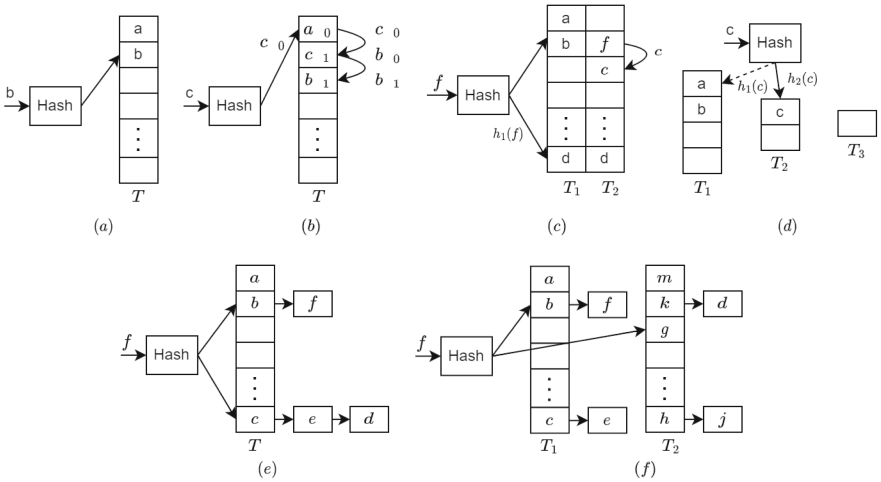
There are many different techniques to minimize the effect of hash collisions, such as chaining, linear/quadratic probing, and double hashing. Chaining is the process where collided items are stored as linked lists, and linear probing searches through the locations in the table sequentially to find an empty slot. Double hashing uses two hash functions where the second hash function is used as an offset to probe for an empty slot in case of a collision. However, techniques such as chaining and linear probing are not really suitable on hardware as the size of the table can increase indefinitely. Also, these techniques worsen the time complexity of hash tables as it may be required to probe over all of the inserted elements in the worst case scenario.

Robin Hood Hashing: Robin hood hashing [6] is a hashing technique to compensate collisions in hash tables. The principle of Robin hood hashing is to keep the keys which are subjected to collisions close to the originally hashed slots, as shown in Fig. 3(b). It uses probe sequence lengths (PSL) to find a slot during insertion. PSL is the number of probes made by an item before it finds a slot, and the PSL has to be stored along with the key. During a collision, probing starts and if a key in the non-empty slot has a lower PSL it is swapped with the key to be inserted. This probing continues until an empty slot is found for the swapped keys.

The most prominent variants of Robin hood hashing are the Quotient filter (QF) [3] and the Counting Quotient Filter (CQF) [26]. QF is practically a linear probing hash table to reduce collisions and QF does not store the whole key but a short fingerprint of the key in a chained manner. Both QF and CQF support deletion and resizing and both exhibit better cache locality. Each fingerprint is divided into quotient and remainder where the remainder is stored in the

location indexed by the quotient. The collisions cause the stored remainders to be shifted to the subsequent slots linearly. In, CQF some of the remaining slots are dedicated for counters, which improve the performance of QFs on skewed datasets. Similar to hash tables, linear probing in quotient filters makes it unsuitable for hardware because of the complexity of implementation, dynamic resizing, and unpredictability of the number of cycles required for insertion. Shifting the elements in a linear fashion in memory is also not preferred on hardware as only one location can be read/write in a single cycle. Moreover, the performance declines as the occupancy of the QF becomes high, especially after 60% occupancy.

***d*-Choice Hashing and *d*-Left Hashing:** Chained hash tables embrace a completely random approach to find a bucket and links the collided keys to the same bucket which could adversely affect the update/query time complexity as the length of the chain in a random bucket becomes longer. *d*-choice hashing [29] introduced load balancing in the hash table of size m by applying d hash functions and the key is inserted into the bucket which has the lowest load. At the instance of a tie when all the hashed buckets have the same load, a bucket is chosen randomly for insertion. *d*-left hashing [24] improved the load balancing by using d separate hash tables with each table is associated with a single hash function. In 2-left hashing there are 2 tables of size $m/2$ and the key is hashed to both the tables and the key is inserted to the left table if the hashed bucket in the left table has the lowest load or if there is a tie. Compared to *d*-choice hashing, the occurrence of ties is less in *d*-left hashing as the left hash table always has equal or more load compared to the right table. A graphical representation of 2-choice hashing and 2-left hashing is shown in Fig. 3(e) and Fig. 3(f) respectively.



(a) Conventional hashing, (b) Robin hood hashing, (c) Cuckoo hashing, (d) Peacock hashing, (e) 2-choice hashing, (f) 2-left hashing

Fig. 3. Various hashing schemes

Cuckoo Hashing: The issue with most of the hash tables is the worst case query time when enabling collision avoidance techniques. In linear probing and double hashing, the worst case lookup time is $O(log\ n)$, where n is the number of elements inserted. In chained hashing, the worst case lookup time is $O(log\ n/log\ log\ n)$, whereas in two way chaining it is $O(log\ log\ n)$. The best way to eliminate such long lookup times is to use perfect hashing where there are no collisions, but that is more of a hypothetical scenario. A hashing technique that can provide a worst case lookup time of $O(1)$ is Cuckoo hashing [25]. Cuckoo hashing-based data structures provide better collision resistance than hash tables while offering a worst case lookup time of $O(1)$.

A Cuckoo hash table (CHT) uses two tables and two hash functions and each key is stored in only one both tables. A graphical representation of CHT is shown in Fig. 3(c). In case of collisions during insertion, the already existing key in the table is swapped with the incoming key and the swapped key is hashed again and relocates to the hash indexed location if empty. If the relocated index is not empty, this process of kicking out the existing keys continues until it reaches a maximum allocated loop value. If the maximum allocated loop value is reached and insertion fails, the CHT needs to be resized and all the elements should be rehashed with a new hash function. CHT provides a faster query time and better space-occupancy, but the memory requirement is still high as it is required to store the full key in the table. One way to overcome this issue is partial-key Cuckoo hashing, which is the basic principle of Cuckoo filters. A more detailed description of partial-key Cuckoo hashing is given in Sect. 2.3.

Peacock Hashing: Peacock hash table (PHT) is another form of linked/chained hash tables. Peacock hashing [19] is probably the best hardware-friendly solution to chained hash tables. PHT employs multiple hash tables where there is a main table followed by multiple backup tables. The size of subsequent backup tables is scaled down by a scaling factor. All the tables use different hash functions. A graphical representation of PHT is shown in Fig. 3(d). The incoming keys are directed to the larger tables first and if there are collisions, the keys are sent to backup tables. If a predefined probing value is reached and still no empty slot is found, the key is discarded. In order to make the querying faster, a fast filter-preferably a Bloom filter-is associated with each of the backup tables in the on-chip memory and the hash tables are stored in the off-chip memory. Despite being a chained hash table, the number of probes due to collision is limited to the number of tables. The worst case update and query time complexity is $O(log\ log\ m)$, where m is the size of the main table.

2.2 Bloom Filter and Its Variants

A Bloom filter [4] is a space-efficient probabilistic data structure which is commonly employed to perform constant-time membership queries in a set S of n elements, $\{x_1, x_2, ..., x_n\}$, such that $S \subseteq U$ where U is a universal set. Proposed by Howard Bloom in 1970, standard Bloom filters (SBF) became an integral part of most of the applications where conventional membership queries turned out to be impractical given that the amount of data to be handled is large. A

Bloom filter is composed of a single-bit array of m bits and all bits are initialized to 0. A simple representation of a SBF is shown in Fig. 4. A set of k independent hash functions are used to map an input x to k locations.

Compared to hash tables, the use of multiple hash functions in Bloom filters somewhat eliminates the requirement of hash-collision avoidance mechanisms. To insert a new element into the filter, all the bits in the k hash-indexed locations are set to 1. While querying for x, the Bloom filter returns $x \in S$ if all the k hash-indexed locations return 1. If any one of the bits in the k hashed locations are not 1, the filter returns $x \notin S$. Bloom filters are simple, easy to construct, and memory-efficient, but still have some limitations such as no support for deletion. Many different variants of Bloom filters have been proposed to address these limitations [13]. There exist more than 50 variants of Bloom filters and a lot more optimized Bloom filter architectures, but there is always a trade-off between memory, accuracy, and speed among these variants and most of the variants are a workaround to eliminate the limitations of Bloom filters.

Support for Deletion: As Bloom filters cannot support deletion, Counting Bloom filter (CBF) [11] is introduced to enable deletions in a BF, but at the cost of a higher space utilization. A CBF follows the same structure of SBF but each single-bit slot in the BF is replaced by a counter to keep track of insertions. Whenever an item is inserted, the hash-indexed counters are incremented by one and during deletion the corresponding counters are decremented by one. Numerous variants and optimizations to CBF have been proposed in recent times to improve CBF. Deletable Bloom filter (DIBF) [30] tries to address the higher memory requirement of CBF while offering the support for deletion. DIBF divides the bit array of size m into r regions and keeps a bitmap of size r-bits to encode whether or not a region is collision free. Each bit in the bitmap represents the collision status of each region. However the trade-off is a higher FPR as the size of each region is a small fraction of m. Moreover, the deletion becomes impossible when every region has at least one collision and DIBF acts like a normal BF with a worsened FPR. Spectral Bloom filters [8] and Space-code Bloom filters [18] also support deletions by following a similar approach as CBF, but targets multi-sets.

Dynamicity: SBF supports unlimited insertions at the cost of a higher FPR, but does not support dynamic resizing where it resizes the existing filter on the run while retaining the same FPR. Dynamic Bloom filter (DBF) [16] and Scalable Bloom filter (SCBF) [2] propose dynamic resizing of the filter adapting to dynamic datasets. Both dynamic and scalable Bloom filters follow the same data structure which consists of a series of small SBFs appended sequentially and the difference being the sizes of the incremental SBFs. DBF has the same size m for all the SBFs whereas the i^{th} SBF of SCBF has a size equal to $m \times a^{i-1}$, where a is a positive integer. Both approaches are slower and have a lower FPR for the same amount of memory, compared to SBF. SCBF has a lower FPR compared to DBF while DBF is faster than SCBF as it uses homogeneous SBFs while SCBF employs heterogeneous SBFs.

Enhanced FPR: SBF, being a simple data structure which enhances the FPR without a trade-off, is difficult. Retouched Bloom filter (RBF) [9] improves the FPR of the Bloom filter by trading for some false negatives. The removal of false positives in RBF is achieved by clearing the corresponding bits.

Speed Optimizations: The larger size of a Bloom filter compared to the size of the cache line, causes cache misses. This issue along with the poor data locality of Bloom filters is addressed by Blocked Bloom filter (BBF) [27]. A BBF has b small sized standard Bloom filters, each of which has a size less than or equal to the cache line. An SBF block is chosen using a hash function and each item is mapped to that SBF using k hash values. This improves the speed, but at the cost of a higher false positive rate as a small single SBF can be filled quickly. This, in fact, results in a need to increase the size of the filter. The optimization in BBF is to improve the run-time performance on a hardware platform, but not a hardware-oriented design that may leverage the same performance when translated on to hardware. Another technique to enhance the speed is to reduce the hash computations. Through double-hashing Bloom filter, Kirsch et al. [17] have shown that only two independent hash functions are enough to generate all the required hash values in the form of $h_1(x) + i * h_2(x)$ without any increase in the asymptotic FPR, where $h_1(x)$ and $h_2(x)$ are hash values of the item x and i is an arbitrary value.

Some of the optimizations on Bloom filters focus on reducing the latency to a single memory access cycle in the likes of Bloom-1 [28] and Parallel Bloom filter (PBF) [32]. Bloom-1 uses a memory array of size m where each location contains a membership word of 32 or 64 bits and an item is mapped to any one of the membership word using k hash functions. Bloom-1 has a lower false positive rate than SBF because of a smaller of the membership word size and require more hardware resources. PBF on, the other hand, is faster and requires less hash bits compared to SBF. PBF splits the memory block of size m into k sub-blocks of size m/k called a Uni-SBF and each memory block is a single hash function. The FPR of a Uni-SBF is $1 - e^{-kn/m}$ and the FPR of the PBF is $(fpr(Uni\text{-}SBF))^k$ which is equal to the FPR of the SBF. A representation of PBF is shown in Fig. 4. PBF is able to achieve an update and query complexity of $O(1)$ compared to a complexity $O(k)$ of SBF.

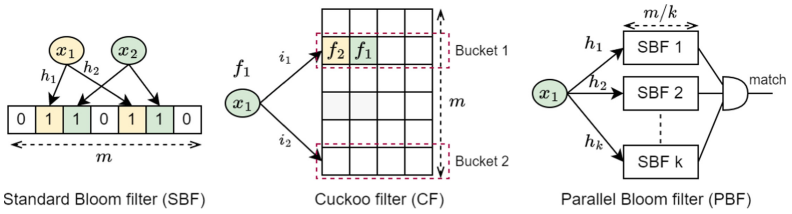


Fig. 4. Bloom and Cuckoo filters

2.3 Cuckoo Filter and Its Variants

A cuckoo hash table stores the keys in the hash-indexed buckets which contributes to a larger memory footprint when the key size is large. When it is required to perform processing at line-rate, constraining the lookup architectures within the on-chip memory is of utmost importance. Partial key Cuckoo hashing [21] helps to resolve this issue by storing only a fingerprint of the key. Cuckoo filter (CF) [10] is based on partial key cuckoo hashing which is very much similar to cuckoo hashing but instead of storing a full key, only a fingerprint/short digest of the key is stored. Figure 4 depicts a representation of the Cuckoo filter. It is composed of a memory block having m locations where each location is termed as a bucket which is indexed by hash value. Each bucket is having b entries and b is set to 4 which provides the best space-efficiency. Every item mapped to two buckets. The indices of the bucket B_i for each item x_i are i_1 and i_2, where $i_1 = hash(x_i)$ and $i_2 = i_1 \oplus hash(f)$, where f is the fingerprint of x_i and is generated using another hash function. During an insertion, if either bucket B_1 or B_2 has an empty slot, the item is inserted to that slot. If none of the slots are empty, a bucket and entry is chosen randomly and the existing item is swapped with the incoming item. This process of swapping the item is called kicking. The bucket index of the swapped item f_s is then computed using $i \oplus hash(f_s)$, where i is the existing location of f_s. f_s is then tried to add to the new slot and if that slot is not empty, this process of kicking continues until it finds a new slot or the maximum value of probing is reached. Cuckoo filter can deny an insertion if either the table is full or the maximum loop value of kicking is reached. In such cases resizing and rehashing is required which is not feasible when the processing is at line-rate.

Compared to Bloom filters, Cuckoo filter is faster, more space efficient, and supports deletions. Irrespective of all the advantages, there are some drawbacks which are very critical when implementing on hardware. Other than the complexity of implementation on hardware, the insertion length is indefinite because of kicking as the load factor of the filter increases. The number of kicks could reach the maximum value of the loop, which is set to an arbitrary value of 500 by Fan et al. [10]. Another disadvantage is the insertion limit where no more insertions are possible once the filter is full. Insertion of duplicate items can affect adversely on the false positive rate and will limit the number of unique items that can be entered into the filter. However, denying the insertion of duplicates is not feasible because two different items with the same fingerprint can be mapped to a single bucket due to collisions and denying the insertion would cause removal of the only existing fingerprint during deletion.

Many different variants of Cuckoo filters have been proposed to enhance the performance of Cuckoo filters on various aspects such as space efficiency, FPR, and speed. Nevertheless, there is always a trade-off between these aspects more often than not. Some of the important variants which are worth mentioning are included here.

Enhancing Speed: Morton filter [5] is a compressed version of Cuckoo filter (CF) in which the storage of data is more dense. Morton filter is able to achieve

an improved memory access time by efficiently utilizing the cache which also helps to achieve a higher insertion, lookup and deletion throughput on ARM architecture compared to CF. However, reduced support to various fingerprint sizes limits the application range of Morton filter. Vertical CF [12] also reduces the insertion time of the CF by increasing the number of buckets for each item. Vacuum filter [35] is also more space efficient and faster than Bloom and Cuckoo filters while achieving the same false positive rate (FPR) as CF. Vacuum filters follow the same data structure as CF but with better data locality which is achieved by dividing the table into multiple chunks similar to Blocked BF and with two different insertion algorithms based on the number of entries to be stored. This division also helps to keep the table size not a power of two in contrast to Cuckoo filter. This is efficient when the number of elements to be stored is not a power of two. Nevertheless, keeping the size of the filter non-powers of two makes things more difficult on hardware.

Enhancing FPR: Adaptive CF [23] improves the FPR of CF by removing the false positives that already occurred. This halts the repeating occurrence of the same false positives. Length-aware CF [20] is also able to reduce the FPR but with added storage requirements. D-ary CF [36] improves the space utilization with a sacrifice on the insertion, deletion, and query performances. D-ary CF uses d hash functions and can achieve a better FPR for the same amount of memory compared to CF, but a decline in speed makes it unfavourable to be a better replacement of CF.

Dynamicity: Similar to BF, various approaches were proposed to enable dynamic resizing in CF. Dynamic CF (DCF) [7] appends multiple homogeneous CFs together similar to dynamic Bloom filters when it is required to extend the size of the filter. Moreover, it can merge the under-utilized CFs to further optimize the space utilization. While DCF can support dynamic resizing, the lookup performance is worse compared to CF as it is required to access multiple linked CFs. Consistent CF (CCF) [22] is a further improved variant of dynamic CF, where each CCF is composed by attaching multiple index-independent Cuckoo filter (I2CF), where each I2CF can have k buckets and the value of k is a variable depending on the cardinality. The sparse I2CF can also be compressed. Nevertheless, dynamic resizing is applicable only for multi-sets and makes the implementation task more cumbersome on hardware.

Other Filters Which are More Space Efficient than Cuckoo Filters: Xor filter [14] and Binary fuse filter [15] offer smaller memory footprint than Cuckoo filters. The update time of Xor filters and Binary fuse filters is more than Bloom and cuckoo filter, but has a faster query time. However, these filters are immutable, which means that dynamic or in-line updates are not possible, which makes it ill-suitable for streaming applications. In order to update a new set of keys, the filter has to be rebuilt and the full set of keys is required at the time of construction.

To conclude this brief description of these data structures, a summary of the time complexities and FPR of important data structures are shown in Table 2.

We can infer from the table that the best-case update/query complexity of structures derived from hash table and cuckoo filter are $O(1)$, and the worst-case update/query could increase as the table is filling up. In contrast, Bloom filter based data structures have a constant update/query complexities irrespective of the load.

3 Hardware Architectures

3.1 Choosing a Suitable Architecture

The requirements of lookup and key-value stores are not the same for every application. This has to be taken into consideration while choosing the best data structures. Our main focus is on network applications where the cardinality is predictable and the size of the key/flow ID is constant throughout. The most suitable data structures from the existing ones are filtered, based on the above said criteria for hardware evaluation.

Table 2. FPR and time complexities of data structures

Datastructure	False positive rate	Time complexity (Update)		Time complexity (Query)	
		Best case	Worst case	Best case	Worst case
Hash table	NA	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Cuckoo Hashing	NA	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Peacock hashing	NA	$O(1)$	$O(log\ log\ m)$	$O(1)$	$O(log\ log\ m)$
2-choice hashing	NA	$O(1)$	$O(log\ log\ n)$	$O(1)$	$O(log\ log\ n)$
Quotient Filter	$1/2^{r}*$	$O(1)$	$O(log\ m)$	$O(1)$	$O(log\ m)$
Bloom Filter	$(1-e^{-k.n/m})^{k}$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Counting Bloom Filter	$(1-e^{-k.n/m})^{k}$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Dynamic Bloom Filter	$1-(1-e^{-k(n-c\lfloor n\rfloor/c)/m})^{k}$	$O(k)$	$O(k)$	$O(k.s)$	$O(k.s)$
Deletable Bloom Filter	$(1-(1-(1/(m-r)))^{k*n})^{k}$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Parallel Bloom Filter	$(1-e^{-k.n/m})^{k}$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Bloom-1 Filter	Refer [31]	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Retouched Bloom filter	$(1-e^{-k.n/m})^{k}*(1-z/p_1.m)^{k}$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Double-hashing BF	$(1-e^{-k.n/m})^{k}$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Cuckoo Filter	$2^{-(C*\alpha-2)}$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Morton filter	$1-(1-1/2^{f})^{\alpha_L.B.S}$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Vacuum filter	$2b\alpha/2^{f}$	$O(1)$	NA	$O(1)$	$O(1)$
D-ary CF	$k/2^{f}$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Dynamic Cuckoo filter	$2.b.s/2^{f}$	$O(1)$	$O(n)$	$O(1)$	$O(2.b.s)$
Consistent CF	$s.k.b/2^{f}$	$O(1)$	$O(N.log\ m)$	$O(1)$	$O(k.b.s.log\ m)$

$m=$ No. of buckets; $n=$ No. of items; $f=$ Fingerprint size; $k=$ No. of hash functions; $b=$ No. of entries in a bucket; $s=$ No. of filters
$c=$ capacity of a single BF; $r=$ No. of regions in the BF; $p_1=1-e^{-kn/m}$; $z=$ No. of bits reset in BF; $\alpha_L=$ logical load factor
$\alpha=$ load factor; $C=$ Bits per item; $B=$ buckets accessed per negative lookup; $S=$ logical slots per bucket; $N=$ max probes allowed;

Lookups: For lookups, where the requirement is only the presence of an item, Hash tables and hash table variants are probably a luxury in terms of memory requirement and implementation complexity. Hash tables are required to

store the key which consumes a large amount of memory when the cardinality is large. Moreover, the requirement of collision resistant mechanisms makes it slower where fast lookup is a necessity when considering processing at line-rate. With inherent collision resistant mechanisms, Bloom filters and Cuckoo filters along with their numerous variants offers the best possible accuracy within the lowest memory requirement.

Key-Value Store: For applications where getting the presence of an element is not enough and either the key or value has to be stored and/or returned, Bloom and Cuckoo filters are limited. Bloom filters do not store keys or values and recovering the key from a Bloom filter is impossible. Cuckoo filters store a fingerprint, but the possibility of recovering the key from the fingerprint is also zero. The case is similar to all the data structures which store only a hashed digest. This makes hash tables and its variants more suitable for probabilistic key-value stores. Nevertheless, hash tables must be associated with collision resistant mechanisms. Looking from a hardware perspective, hardware suitability of mechanisms such as chaining and linear probing is low. Probing in the form of Peacock hashing and Cuckoo hashing are more hardware friendly as the size of the table is fixed and are suitable for static data sets.

3.2 Implementation Details

Optimized Hashing. Hashing is one of the most important building block of AMQ data structures as the overall throughput of the system can be affected by the speed of the hash computation. Non-cryptographic hashes with satisfactory avalanche properties are preferred as it is faster and has a low logical depth compared to cryptograhic hashes. Work by Sateesan et al. [31] proposed a fast non-cryptographic hash function Xoodoo-NC, which is derived from the Xoodoo permutation. In this work, Xoodoo-NC is used to generate the required hash bits and these hash bits are then split into required hash values. Xoodoo-NC can generate outputs as multiples of 96-bits. Recent works [28,32] have shown that this method of splitting the hash output to generate Bloom filter hash values has negligible effect on the false positive rate (FPR) of the filter and can achieve multi-fold improvement in latency.

Even though the key size is fixed to 96-bit in our evaluations, a varying key-size will only have a negligible effect on the AMQ algorithms. The FPR is not affected by the size of the input as observed from Table 2 as long as the hash function satisfies the required avalanche properties.

Hash Table. The representation of the hash table (HT) is shown in Fig. 3(a). The implementation of hash table is very straightforward and is implemented using block RAMS (BRAM) as a memory block having depth m and width equal to the size of the key and value. In a naive, straightforward implementation of the hash table, collisions are not addressed, and is implemented as a basic unit for comparisons.

Cuckoo Hash Table. A representation of the Cuckoo hash table (CHT) is shown in Fig. 3(c). Unlike in the figure, the tables T_i and T_2 are implemented as separate BRAM blocks where each memory block is having one entry of each bucket. Such an implementation halves the word-length of one bucket which is equal to $2 * (size\ of\ the\ key + size\ of\ the\ value)$. Since both the memories can be accessed in parallel, the latency to access a bucket is still a single clock cycle.

Peacock Hash Table. To implement Peacock hash table (PHT), the main table and all the backup tables as shown in Fig. 3 are implemented as separate BRAM blocks. The size of the main table is m and scaling factor $r = 2$, which the is most appropriate value of r for hardware. Since our goal is to analyze the performance of only the hash table, no fast lookup mechanisms are employed as presented by Kumar et al. [19]. PHT has $(1 + log_{1/r}m)$ tables and the last table has only one location. The last table is eliminated in the implementation as it is not possible to generate a BRAM block with only 1 location. All memories are accessed in parallel to keep the update/query latency to a bare minimum unlike the original algorithm in which the i^{th} table is accessed only when the results are not found with the $(i-1)^{th}$ table.

Bloom Filter and Parallel Bloom Filter. Standard Bloom filter (SBF) is straightforward and implementation is hassle-free. The data structure as depicted in Fig. 4, is implemented with a BRAM block of depth m and width 1-bit. The number of hash functions k is 8 and these hashes are generated using Xoodoo-NC. SBF with 8 hash functions requires a total of 136 hash bits, and Xoodoo-NC generates an 192-bit output and this output is split into 8 hash values of sizes $log_2 m$ bits each. For parallel Bloom filter (PBF), all the k sub-blocks of size m/k are implemented separately using BRAM. Each sub-block is accessed in parallel, which keeps the memory access latency to a single cycle. Xoodoo-NC is used to generate k hash values for PBF in a similar way as it is for SBF.

Cuckoo Filter. In contrast to Bloom filter, implementing cuckoo filter on hardware requires a bit more effort and engineering. The data structure of CF is shown in Fig. 4. The whole table of CF is implemented as a single BRAM block. The depth of the memory is m and the width of the memory is $b * f$, where f is the size of the fingerprint and b is the number of buckets. The memory is implemented as true dual port RAM with dedicated ports for reading and writing. It requires two clock cycles to read the contents in both the buckets. CF is optimized for the number of memory accesses in such a way that one bucket is read at first in parallel with the computation of the address of the second bucket and the second bucket is read only if no empty slots are found in the first bucket during an update. Similar search is applied during query also, which makes the best case memory accesses to a single cycle both during update and query.

The hardware architecture diagrams of all the implemented data structures except the Bloom filter architectures are presented in Appendix A. The architectures presented in [32] are followed for Bloom filter implementations.

4 Evaluation

The evaluation is performed separately for lookup and key-value store data structures. The analysis is performed in terms of FPR or accuracy, latency, hardware resource usage, speed, insertion throughput, and implementation complexity. Some additional analysis on throughput, memory access cycles and insertion failures are presented in Appendix B.

4.1 Evaluation of Lookup Architectures

For the evaluation, the architectures chosen are Cuckoo filter (CF), standard Bloom filter (SBF), and Parallel Bloom filter (PBF). A total of 16 KB memory is allocated to all the data structures and the size of the tables of each structure is determined based on the allocated memory. The number of bits per item is fixed to 12, the number of items n to be inserted equals 10922. This value of n makes the load factor α of CF to be 0.67 and $m = 4096$. The CF consists of 2 buckets and each bucket is having $b = 4$ entries and the fingerprint size is set to 8-bits. The maximum kick value is set as 500, which is the optimal value employed in the original article [10]. In the evaluation, memory access cycles refers to the sum of memory read and write cycles. The number of hash functions for Bloom filters are $k = 8$ which makes $m = 131{,}072$ for SBF and PBF. Size of a single block of PBF is $m/k = 16384$. The hardware evaluation is performed using synthetic datasets on a Virtex UltraScale+ (xcvu9p-flga2104-2L-e) platform.

False Positive Rate and Space-Occupancy. For a fixed bits/item, the FPR ϵ of CF is better than Bloom filters as shown in Table 3, which means better space-occupancy. CF can have 2*b duplicate entries, where b is the number of entries in a bucket. In order to store similar number of duplicates, a CBF would require a 3-bit counter in each location which results in 3 times more memory requirement compared to SBF. However, allowing duplicates will deplete the space and cause higher false positives for CF, which does not happen with SBF.

Latency. The query time complexity of CF is $O(1)$ and of SBF and CBF is $O(k)$. However, PBF outperforms CF in all other aspects, except in terms of number of hash bits and FPR for a fixed number of bits per item. Both the update and query time complexity of PBF is $O(1)$, thanks to the parallel accessing of memory blocks. Memory access cycles to insert an element is constant for Bloom filters irrespective of the load factor. CF consumes more memory access cycles for insertion due to probing/kicking when the filter starts filling up. The analysis given on Table 3 shows that the percentage of memory access cycles due to probing is only around 5.7% more than the actual requirement if the load factor is 75%. However, probing memory accesses increases to 14.2% when load factor is 85% and then a sudden spike to 48.2% when the load factor is 95.5%. The total number of memory access cycles required for PBF is less than that of CF for a load factor up to 85%, and is only less than half than that of CF when the load factor is 95.5%.

Table 3. FPR and latency vs various load factors for lookup data structures

Load factor	False positve rate		Total memory access cycles			Total probe cycles (CF)	% of cycles for probing (CF)
	CF	SBF, PBF	SBF	PBF	CF		
35%	$9.8x10^{-7}$	$5.5x10^{-5}$	91,200	11,400	11,480	6	0.05%
50%	$1.0x10^{-4}$	$6.0x10^{-4}$	131,200	16,400	16,801	72	0.43%
65%	$1.1x10^{-3}$	$2.7x10^{-3}$	170,400	21,300	22,709	500	2.20%
75%	$3.4x10^{-3}$	$6.0x10^{-3}$	196,560	24,570	27,699	1,568	5.66%
85%	$7.8x10^{-3}$	$1.1x10^{-2}$	222,800	27,850	35,361	5,036	14.24%
95.5%	$1.6x10^{-2}$	$2.0x10^{-2}$	250,320	31,290	67,658	32,584	48.16%

Average of 1000 runs; Total memory = 16KB, # of items = 16384 (at $\alpha = 1.0$)
CF-Cuckoo filter, SBF-Standard Bloom filter, PBF-Parallel Bloom filter

Performance on Hardware. The performance results on FPGA are shown in Table 4. The hardware resource requirements and maximum operating frequency of SBF and PBF are better than CF while maintaining the same number of bits per item, but with a lower FPR and more hash bits. CF requires more than 2 times the number of LUTs compared to the Bloom filter counterparts. CF requires 2 cycles for hashing as the second hash computation is dependant on the first. Nevertheless, CF can still achieve a best case query latency of 2 cycles if the memory read of one bucket can be performed in parallel while computing the second hash index. Since the number of probes during insertion can vary up to 500 as α increases, pipelining becomes difficult. PBF has a constant query latency of only 2 cycles (1 cycle for hashing, 1 cycle for memory read) irrespective of the load. Even though SBF is the simplest to implement on hardware, it requires 9 cycles for querying. In terms of throughput, PBF has an edge over CF and delivers the best insertion throughput while BF and CBF has a very low throughput due to its low latency.

Table 4. Performance of Lookup architectures on FPGA

	Cuckoo filter (CF)	Bloom filter (SBF)	Counting Bloomfilter (CBF)	Parallel Bloom filter (PBF)
Load Factor (α)	0.67	–	–	–
FPR (ϵ)	**0.0015**	0.0031	0.3124	0.0031
# of hash bits	**24**	136	136	112
Best case query Latency	**2 cycles**	9 cycles	9 cycles	**2 cycles**
LUT	930	402	412	**336**
FF	267	259	288	259
BRAM	4	4	4.5	4
Max. frequency	435 MHz	476 MHz	435 MHz	**488 MHz**
Insertions/second (Million)	138	28	26	**163**
Implementation complexity	++++	+	++	++

Memory = 16KB, # of items = 10922 (at $\alpha = 0.67$), bits per item = 12

Discussion. Even though CF has some clear advantages over SBF and most of the other variants of Bloom filter (BF), the achievable parallelism in BF helps to generate better results on hardware. PBF helps to achieve an update and query memory access latency of a single clock cycle. While BF has no insertion limit, CF can deny an entry if the number of kicks exceeds the maximum kick value. Incrementing the size of the filter or re-hashing all the elements (when it comes to cuckoo hashing) dynamically is a cumbersome task in hardware, especially for online processing. Duplicate entries have no effect on BF, but drastically worsens the load factor, space occupancy, and FPR of CF. The maximum number of duplicates that CF can accommodate is $2*b$, where b is the number of entries in a bucket. Trying to insert duplicates after $2*b$ times would result in a infinite kicking loop until the max kick length is reached. In network security applications, an attacker can exploit this vulnerability of CF. Deletion support is one of the prominent feature of CF which BF cannot provide. CBF can provide deletion support, but with much higher memory usage. In conlusion, if the choice is for low-latency, lightweight lookup architecture on hardware with the support for pipelining, PBF is the best choice over CF. For a better bits per item with deletion support, where pipelining is not a primary requirement and given that the insertion of duplicates is minimal, CF is the preferred option.

4.2 Evaluation of Key-Value Stores

For the evaluation, Conventional hash table (HT), Cuckoo hash table (CHT), and Peacock hash table (PHT) are compared as discussed in Sect. 3.1. The memory allotted are fixed to 224KB for each data structure. CHT has two tables and each table has a size $m = 8192$ buckets and each bucket has 1 entry each per table. The memory accesses are optimized for CHT similar to the optimizations in CF as discussed in Sect. 4.1. The maximum kick value is set to 500. The size m of the main table in PHT is set as 8192 and the scaling factor $r = 0.5$. The total number of tables t in PHT is 13 with each i^{th} backup table having a size of $m/(\frac{1}{r})^i$ where $1 \leq i \leq t-1$. HT has a size m of 16384 buckets. The key size is 96-bits and value size is 16-bits. The load factors of CHT is $n/2m$, PHT is $m(1-r^t)/(1-r)$, and HT is n/m. In the evaluation, accuracy refers to the ratio of number of correctly queried items from the table and the total number of items inserted to the table.

Loading the Table and Accuracy The accuracy and probing cycles in hash tables increase with increasing load factor α as shown in Table 5. In HT, the probability of failure in inserting an item is high as no collision resistant mechanisms are employed. PHT has an accuracy close to 100% when $\alpha = 0.6$ and the accuracy and number of failed insertions worsens as $\alpha > 0.6$. For CHT, the accuracy is close to 100% even at $\alpha = 0.88$. The accuracy of CHT can be improved by increasing the number of entries in each bucket. Then, the table depth must be reduced accordingly to keep the memory requirement constant.

Table 5. FPR and latency vs various load factors for key-value stores

Load factor	Accuracy			Total memory access cycles			% of cycles for probing	
	HT(%)	PHT(%)	CHT(%)	HT	PHT	CHT	PHT	CHT
35%	84.5	100.0	100.0	11,468	13,635	12,052	18.9	1.9
50%	78.6	100.0	100.0	16,384	21,246	18,751	29.7	7.5
65%	73.6	99.5	100.0	21,300	30,959	29,720	45.3	21.8
75%	70.3	97.4	100.0	24,576	39,928	45,909	62.5	40.3
85%	67.3	93.7	100.0	27,852	51,518	106,891	84.9	70.2
90%	65.9	91.3	99.3	29,492	58,719	328,079	99.1	89.6
95%	64.5	88.9	96.4	31,130	66,185	847,033	112.6	95.8

Average of 1000 runs; Total memory = 16KB, # of items = 16384 (at $\alpha = 1.0$)
HT-Hash table, PHT-Peacock hash table, CHT-Cuckoo hash table

Latency CHT has a worst case query complexity of $O(1)$, whereas the query complexities of other hash tables vary with the chain/probe lengths. CHT has an average update time complexity of $O(1)$, but the probing length increases as α increases. When $\alpha > 0.5$, the number of probes increases drastically for CHT and can go up to a maximum probe length of 500 as set. PHT is very much like a chained hash table, but since the table size is fixed in PHT, the maximum probe length is limited to the total number of tables in PHT. The number of memory access cycles also increases for CHT due to probing as α increases. For example, the increase is ≈4417% when the load factor increases from 0.5 to 0.95. The increase in memory access cycles are considerably lower for PHT, which is an increase of ≈211% when the load factor increases from 0.5 to 0.95.

Performance on Hardware. The performance results on FPGA are given in Table 6. All the tables in PHT and CHT can be accessed in parallel, which makes

Table 6. Performance of hash table architectures on FPGA

	HT	CHT	PHT
Accuracy at $\alpha = 0.95$	64.5%	96.4%	88.9%
# of hash bits	14	26	91
Best case Query Latency	2 cycle	2 cycle	2 cycle
LUT	606	1745	1399
FF	268	971	267
BRAM	50	50	79
Max. frequency	357 MHz	322 MHz	232 MHz
Insertions/second (Million)($\alpha = 0.75$)	119	68	42
Implementation complexity	+	++++	++

Memory = 224KB, # of items = 15565 (at $\alpha = 0.95$), bits per item = 118
HT-Hash table, PHT-Peacock hash table, CHT-Cuckoo hash table

the query latency of all three architectures equal to 2 clock cycles which include hashing and memory access cycles. It is evident from the table that there is always a trade-off among space occupancy, speed, and accuracy. Conventional hash table is the least accurate, but it is faster and consumes the least amount of resources. The memory footprint is higher for PHT as the memory is split into many smaller blocks which takes a lot more memory than a large single block. This is because of the fact that the targeted FPGA can construct the memory only with either 18Kb/36Kb block RAM (BRAM) modules. Moreover, these blocks, spread around the main logic, cause the routing delay to increase which eventually causes the maximum operating frequency to dip. One remedy is to keep a limited number of tables, but this would cause the accuracy to decline because of the increased number of insertion fails as the probing during collision is limited by the number of blocks. For example, keeping the total number of tables to 4 and each table having equal sizes of 4096 buckets would cause the accuracy to drop to 46% when the load factor is 1.0 compared to 86% of the conventional PHT. In a way, CHT employs the same principle as PHT which probes for an empty slot during a collision, but with a better space occupancy. However, the complexity of the architecture is high for CHT when implementing on hardware. This results in a higher resource consumption in terms of LUTs. Nevertheless, CHT leverages significantly better operating frequency than PHT. In terms of throughput, CHT has better insertion throughput until $\alpha = 0.75$, but the throughput drops drastically for CHT and is only 6 M items/s when $\alpha = 0.95$ whereas PHT still maintains a throughput equal to 30 M items/s at $\alpha = 0.95$.

Partial Key Cuckoo Hashing for Better Performance. Partial key cuckoo hashing is one way to compromise the high resource and memory requirement of CHT and improve the operating frequency if storing/retrieving the key is not required. This can be implemented by storing the value along with the fingerprints in a CF. However, it is required to have an optimal fingerprint size to mitigate the adverse effect on FPR to some extent. The empirical results show that storing values in a CF, using an 8-bit fingerprint, can leverage the similar accuracy as CHT while CF only uses ≈21% of the memory that is required by CHT. Moreover, CF can run at a significantly higher operating frequency and insertion throughput compared to CHT. Table 7 shows the results of employing CF as a partial key-value store.

Table 7. Results of partial key cuckoo hashing as partial key-value store

Accuracy ($\alpha = 0.95$)	Total cycles	% of extra cycles for probing	LUT	BRAM	Max frequency	Insertions/second (Million)($\alpha = 0.75$)
100%	62,740	44.47%	1442	11	417 MHz	128

Memory = 48KB, # of items = 15565 (at $\alpha = 1.0$), bits per item = 25, Throughput in Million item/s

Discussion. When choosing the best possible key-value store scheme for hardware, PHT might be having a slight edge over CHT in terms of better probing

length at higher load factors and low cardinality. Even though the accuracy is comparatively higher for CHT as observed in Table 5, it comes at a high cost of an extremely large amount of probing cycles when $\alpha > 0.5$. However, as the cardinality increases, implementing peacock hashing becomes hefty as it has to manage a large number of tables, which results in an increased time complexity and much reduced operating frequency. Irrespective of the increased cardinality, query time complexity is always constant with CHT and the operating frequency is also much higher. Hence it can be concluded that PHT is preferable only for a lower cardinality if the criterion is minimal probing cycles whereas CHT can be preferred for any other criteria assuming that the load factor of CHT is kept small. Moreover, if storing/retrieving keys are not required, there is no better alternative than partial key cuckoo hashing to store values.

5 Conclusion

In this paper, various AMQ schemes as well as hash-based probabilistic schemes are analyzed and evaluated based on their hardware-friendliness for network security applications on high-speed networks. A comparison of these schemes is performed in terms of memory efficiency, accuracy, latency, implementation complexity, and throughput. The evaluation results help to identify a suitable data structure for network security applications. Moreover, this analysis also sheds light on the shortcomings of the existing membership query data structures when implemented in hardware, which was unexplored earlier.

Acknowledgement. This work is supported by the ESCALATE project, funded by FWO (G0E0719N) and SNSF (200021L_182005), and by Cybersecurity Research Flanders (VR20192203).

Appendices

A Hardware Architectures

A.1 Cuckoo Hash Table and Cuckoo Filter

The hardware architecture of Cuckoo filter implementation is shown in Fig. 5. Two hash functions, two copies of Xoodoo-NC, are used for generating the hash values. Due to the low logical depth of Xoodoo-NC, there is negligible effect on the overall computation and latency overhead. Cuckoo filter has two buckets with each bucket having four entries. The first hash function is used to generate the address of the first bucket as well as the fingerprint by hashing the incoming key. The hash output is split to generate the required memory addresses and fingerprint. The second hash function is used to generate the address of the second bucket by hashing the fingerprint. The second hash function is re-used for generating the address of the bucket from the fingerprint during kick operations. A multiplexer determines whether the input to the second hash function

is a kicked fingerprint or not. Even though a single hash function is enough to perform all the hashing operations, addition of a second hash function makes pipelining easier and helps to achieve a best case memory access latency of a single clock cycle. The fingerprint is padded with zeroes at the MSB to make a 96-bit input to the hash function. An Finite State Machine (FSM) is employed as the control logic, which controls all the memory read/write operations and the kick operations. The kick logic co-ordinates all the kick operations during the occurrence of a collision. A single dual port BRAM is used as the table and each location in the memory (a bucket) holds four entries.

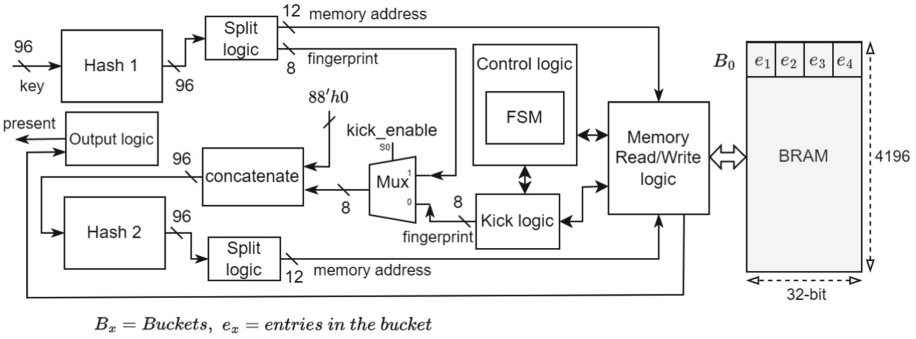


Fig. 5. Hardware architecture of Cuckoo filter

The hardware architecture of Cuckoo hash table (CHT) is very much similar to Cuckoo filter and is shown in Fig. 6. Cuckoo filter employs partial-key cuckoo hashing where a fingerprint of the key is used to generate the second memory address as discussed in Sect. 2.3, whereas CHT hashes the key to generate both addresses. CHT has two buckets and each bucket contains two entries. Since both hash values can be computed in parallel, two copies of Xoodoo-NC and two separate memory blocks are used to access both the buckets in parallel which limits the read/write latency to a single clock cycle. Each memory block has one entry in each bucket, which also helps to limit the word-length of the memory. The second hash function is re-used to generate the hash values during kick operations and a multiplexer determines whether the input to the second hash function is a kicked key or not. All other operations and logic are similar to the Cuckoo filter architecture.

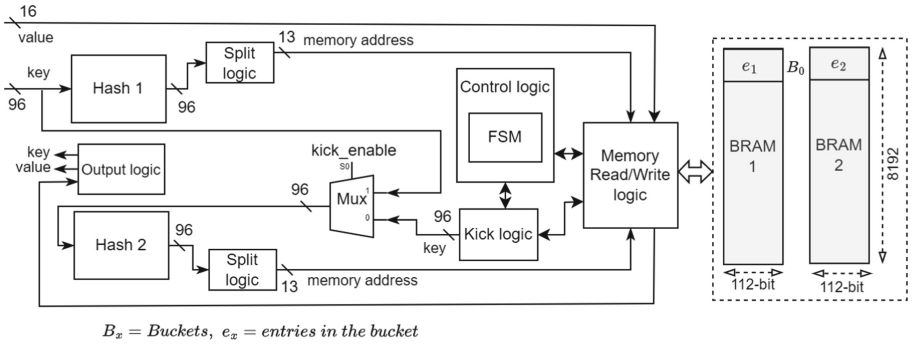


Fig. 6. Hardware architecture of Cuckoo hash table

A.2 Peacock Hash Table

The implementation of Peacock hash table (PHT) is very straightforward and the the hardware architecture is shown in Fig. 7. For an allocated memory size of 224KB, PHT employs a total of thirteen memory blocks as described in Sect. 4.2. In order of minimize the computational as well as latency overhead, a single hash function Xoodoo-NC is used to generate all the required memory addresses. The key is hashed to generate all the required hash bits and a split logic splits the hash output bits to required memory addresses. The probe logic co-ordinates the probing operations during the occurrence of a collision.

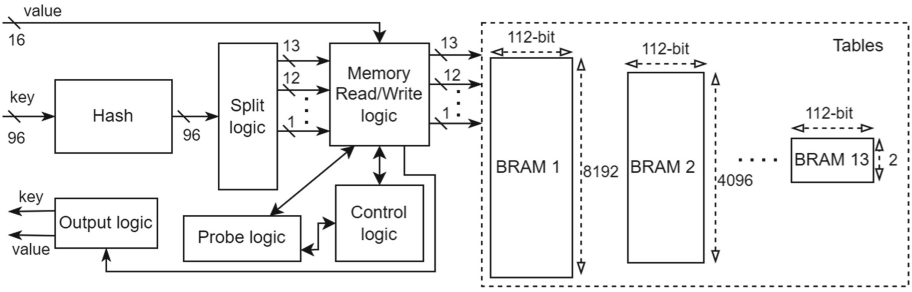


Fig. 7. Hardware architecture of Peacock hash table

B Additional Analysis

B.1 Memory Access Cycles

The total memory access cycles during the insertion of elements for various data structures is depicted in Fig. 8 and 9. Load factor (α) plays an important role when it comes to number of the memory access cycles of hashing-based

data structures such as CHT, PHT, and CF. With increased load factor, the memory access cycles increases for CHT, PHT, and CF because of the kick/probe operations due to collisions. As observed from Fig. 8, there is an exponential increase in the memory access cycles for CHT when $\alpha > 0.75$. Nevertheless, the increase is minimal for PHT because the maximum number of probes is limited by the number of tables. There is a gradual increase in memory access cycles for CF as shown in Fig. 9, but significantly lesser because of the more number of entries per bucket compared to CHT. For Bloom filters, the increase in memory access cycles are constant because of the fact that the number of cycles per insertion is constant throughout irrespective of the load factor.

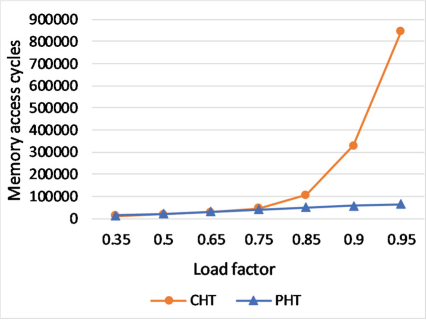


Fig. 8. Memory access cycles for hash tables

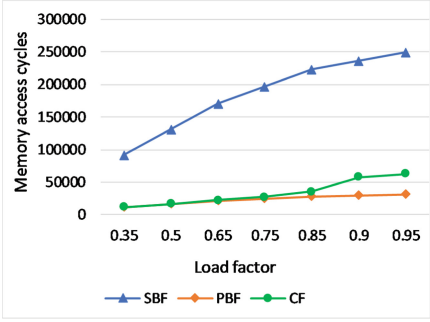
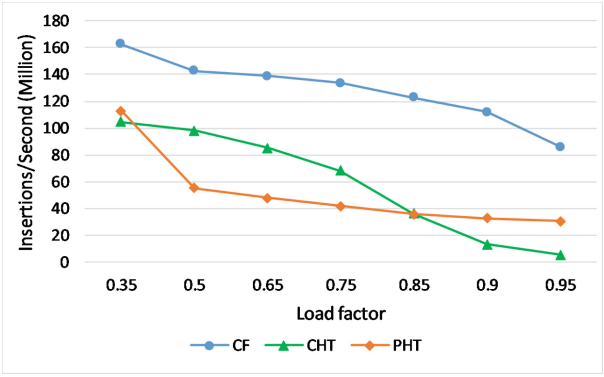


Fig. 9. Memory access cycles of Cuckoo and Bloom filters

B.2 Insertion Throughput

The insertion throughput defines the maximum number of insertions possible per second. The insertion throughput is computed as $\frac{1}{(total\ latency/n))}$, where n is the number of elements inserted and the latency is measured in nanoseconds. The throughput is constant for Bloom filters since the the number of cycles for each insertion is constant. However, the throughput for hashing-based data structures varies with varying load factor because of the extra hash computation and memory access cycles required due to kick/probe operations during a collision. The insertion throughput in terms of million insertions per second of Cuckoo filter (CF), Cuckoo hash table (CHT), and Peacock hash table (PHT) are depicted in Fig. 10.



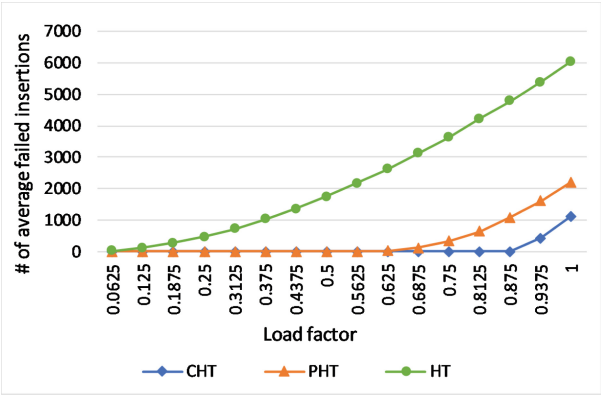
CHT-Cuckoo hash table, PHT-Peacock hash table, CF-Cuckoo filter

Fig. 10. Insertion throughput

CF exhibits significantly higher throughput than CHT, thanks to the higher operating frequency and significantly lesser number of probe cycles compared to CHT. The amount of probing increases drastically for CHT as the load factor increases. Having four entries in each bucket helps to reduce the number of probes for CF, whereas CHT has only two entries in each bucket and this results in a much higher probing cycles compared to CF. Adding 4 entries in a bucket can help to minimize the probing cycles for CHT, but a large memory footprint would still limit the operating frequency. The limited memory footprint of CF contributes to minimal routing delay and hence a higher operating frequency. The throughput of PHT is minimum, even for small load factors, as a result of the lower operating frequency. Yet, the throughput of PHT is almost constant throughout even for higher load factors as the total amount of probing per insertion is limited by the number of tables.

B.3 Insertion Failures

Hash table collisions cause insertion failures while adding elements to the table. Figure 11 shows the insertion failures in hashing-based data structures. It can be seen that insertion failures are very much dependent on the load factor. There are no collision resistant mechanisms such as chaining/probing are employed for the hash table (HT) and it is very much evident from the figure that the collisions are maximum for HT even for lower load factors. When the load factor is 1, the insertion failures for HT is almost 37%. For PHT, the insertion failures start increasing gradually when the load factor is greater than 0.62. For Cuckoo hashing, insertion failures start increasing only after a load factor of 0.87 and the overall failures in CHT is considerably lower than PHT. When the load factor is 1, the insertion failure is around 7% for CHT, whereas it is close to 14% for PHT.



CHT-Cuckoo hash table, PHT-Peacock hash table, HT-Hash table

Fig. 11. Insertion failures in hash tables

References

1. How much data is created every day in 2022? https://earthweb.com/how-much-data-is-created-every-day/. Accessed 25 Jul 2022
2. Almeida, P.S., Baquero, C., Preguiça, N., Hutchison, D.: Scalable bloom filters. Inf. Process. Lett. **101**(6), 255–261 (2007)
3. Bender, M.A., et al.: Don't thrash: how to cache your hash on flash. In: 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11) (2011)
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
5. Breslow, A.D., Jayasena, N.S.: Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. Proc. VLDB Endowment **11**(9), 1041–1055 (2018)
6. Celis, P., Larson, P.A., Munro, J.I.: Robin hood hashing. In: 26th Annual Symposium on Foundations of Computer Science (SFCS 1985), pp. 281–288. IEEE (1985)
7. Chen, H., Liao, L., Jin, H., Wu, J.: The dynamic cuckoo filter. In: 2017 IEEE 25th International Conference on Network Protocols (ICNP), pp. 1–10. IEEE (2017)
8. Cohen, S., Matias, Y.: Spectral bloom filters. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. pp. 241–252 (2003)
9. Donnet, B., Baynat, B., Friedman, T.: Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In: Proceedings of the 2006 ACM CoNEXT Conference, pp. 1–12 (2006)
10. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: practically better than bloom. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, pp. 75–88 (2014)
11. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Networking **8**(3), 281–293 (2000)
12. Fu, P., Luo, L., Li, S., Guo, D., Cheng, G., Zhou, Y.: The vertical cuckoo filters: a family of insertion-friendly sketches for online applications. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS) (2021)

13. Geravand, S., Ahmadi, M.: Bloom filter applications in network security: a state-of-the-art survey. Comput. Netw. **57**(18), 4047–4064 (2013)
14. Graf, T.M., Lemire, D.: Xor filters: faster and smaller than bloom and cuckoo filters. J. Exp. Algorithmics (JEA) **25**, 1–16 (2020)
15. Graf, T.M., Lemire, D.: Binary fuse filters: fast and smaller than XOR filters. J. Exp. Algorithmics (JEA) **27**(1), 1–15 (2022)
16. Guo, D., Wu, J., Chen, H., Yuan, Y., Luo, X.: The dynamic bloom filters. IEEE Trans. Knowl. Data Eng. **22**(1), 120–133 (2009)
17. Kirsch, A., Mitzenmacher, M.: Less hashing, same performance: building a better bloom filter. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 456–467. Springer, Heidelberg (2006). https://doi.org/10.1007/11841036_42
18. Kumar, A., Xu, J., Wang, J.: Space-code bloom filter for efficient per-flow traffic measurement. IEEE J. Sel. Areas Commun. **24**(12), 2327–2339 (2006)
19. Kumar, S., Turner, J., Crowley, P.: Peacock hashing: deterministic and updatable hashing for high performance networking. In: IEEE INFOCOM 2008-The 27th Conference on Computer Communications, pp. 101–105. IEEE (2008)
20. Kwon, M., Reviriego, P., Pontarelli, S.: A length-aware cuckoo filter for faster IP lookup. In: 2016 IEEE Conference on Computer Communications Workshops (2016)
21. Lim, H., Fan, B., Andersen, D.G., Kaminsky, M.: Silt: A memory-efficient, high-performance key-value store. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 1–13 (2011)
22. Luo, L., Guo, D., Rottenstreich, O., Ma, R.T., Luo, X., Ren, B.: The consistent cuckoo filter. In: IEEE INFOCOM 2019, pp. 712–720. IEEE (2019)
23. Mitzenmacher, M., Pontarelli, S., Reviriego, P.: Adaptive cuckoo filters (2020)
24. Mitzenmacher, M.D., Vocking, B.: The asymptotics of selecting the shortest of two, improved (1999)
25. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
26. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: A general-purpose counting filter: Making every bit count. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 775–787 (2017)
27. Putze, F., Sanders, P., Singler, J.: Cache-, hash-and space-efficient bloom filters. In: International Workshop on Experimental and Efficient Algorithms (2007)
28. Qiao, Y., Li, T., Chen, S.: Fast bloom filters and their generalization. IEEE Trans. Parallel Distrib. Syst. **25**(1), 93–103 (2013)
29. Richa, A.W., Mitzenmacher, M., Sitaraman, R.: The power of two random choices: a survey of techniques and results. Comb. Optim. **9**, 255–304 (2001)
30. Rothenberg, C.E., Macapuna, C.A., Verdi, F.L., Magalhaes, M.F.: The deletable bloom filter: a new member of the bloom family. IEEE Commun. Lett. **14**(6), 557–559 (2010)
31. Sateesan, A., Vliegen, J., Daemen, J., Mentens, N.: Novel bloom filter algorithms and architectures for ultra-high-speed network security applications. In: 2020 23rd Euromicro Conference on Digital System Design (DSD), pp. 262–269. IEEE (2020)
32. Sateesan, A., Vliegen, J., Daemen, J., Mentens, N.: Hardware-oriented optimization of bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. Microprocess. Microsyst. **93**, 104619 (2022)
33. Scherrer, S., et al.: Low-rate overuse flow tracer (loft): an efficient and scalable algorithm for detecting overuse flows. In: 2021 40th International Symposium on Reliable Distributed Systems (SRDS). IEEE (2021)
34. Szabo-Wexler, E.: Approximate membership of sets: A survey (2014)

35. Wang, M., Zhou, M.: Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. Proc. VLDB Endowment (2019)
36. Xie, Z., Ding, W., Wang, H., Xiao, Y., Liu, Z.: D-ary cuckoo filter: a space efficient data structure for set membership lookup. In: 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (2017)