SPArch: A Hardware-oriented Sketch-based Architecture for High-speed Network Flow Measurements

ARISH SATEESAN, ES&S-COSIC, ESAT, KU Leuven, Belgium
JO VLIEGEN, ES&S-COSIC, ESAT, KU Leuven, Belgium
SIMON SCHERRER, Department of Computer Science, ETH Zurich, Switzerland
HSU-CHUN HSIAO, National Taiwan University & Academia Sinica, Taiwan
ADRIAN PERRIG, Department of Computer Science, ETH Zurich, Switzerland
NELE MENTENS, ES&S-COSIC, ESAT, KU Leuven, Belgium and LIACS, Leiden University, The Netherlands

Network flow measurement is an integral part of modern high-speed applications for network security and data-stream processing. However, processing at line rate while maintaining the required data structure within the on-chip memory of the hardware platform is a challenging task for measurement algorithms, especially when accuracy is of primary importance, such as in network security applications. Most of the existing measurement algorithms are no exception to such issues when deployed in high-speed networking environments and are also not tailored for efficient hardware implementation. Sketch-based measurement algorithms minimize the memory requirement and are suitable for high-speed networks but possess a low memory-accuracy trade-off and lack the versatility of individual flow mapping. To address these challenges, we present a hardware-friendly data structure named Sketch-based Pseudo-associative array Architecture (SPArch). SPArch is highly accurate and extremely memory-efficient, making it suitable for network flow measurement and security applications. The parallelism in SPArch ensures minimal and constant memory access cycles. Unlike other sketch architectures, SPArch provides the functionality of individual flow mapping similar to associative arrays, and the optimized version of SPArch allows the organization of counters in multiple buckets based on the flow sizes. An in-depth analysis of SPArch is carried out in this paper and implemented SPArch on the Alveo data center accelerator card, demonstrating its suitability for high-speed networks.

CCS Concepts: • **Theory of computation → Sketching and sampling**; **Probabilistic computation**; • **Networks → Network monitoring**; **Network measurement**; • **Security and privacy → Security in hardware**; **Network security**; • **Information systems → Information storage systems**.

Additional Key Words and Phrases: FPGA, Counter array, Probabilistic data structure

Authors' addresses: Arish Sateesan, arish.sateesan@kuleuven.be, ES&S-COSIC, ESAT, KU Leuven, Belgium; Jo Vliegen, jo.vliegen@kuleuven.be, ES&S-COSIC, ESAT, KU Leuven, Belgium; Simon Scherrer, simon.scherrer@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland; Hsu-Chun Hsiao, hchsiao@csie.ntu.edu.tw, National Taiwan University & Academia Sinica, Taiwan; Adrian Perrig, adrian.perrig@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland; Nele Mentens, nele.mentens@kuleuven.be, ES&S-COSIC, ESAT, KU Leuven, Belgium and LIACS, Leiden University, The Netherlands.

1 INTRODUCTION

Network flow measurement plays a crucial role in many network applications, such as large flow detection. A simple task like counting network packets poses a real challenge in such applications when processing data streams at line rate in high-bandwidth networks. Although existing efforts have managed to enhance accuracy and speed on software platforms, they often fail to harness the complete potential for speed improvement offered by hardware platforms. In recent times, Field Programmable Gate Arrays (FPGAs) have been widely employed in data center applications and are proven to be a resilient foundation for high-performance networking applications [5, 24, 36, 37]. The advent of such high-performance computing platforms like FPGAs compels a re-evaluation of measurement architectures to fully exploit parallelization capabilities and achieve maximum throughput. Present algorithms are software-centric and sequential, making it difficult to exploit the parallelism on hardware effectively. Terabit Ethernet networks, networks with speeds exceeding 100 Gigabits per second (Gbps), require immediate online handling of network packets for network security applications, and hardware architectures play an indispensable role in real-time packet processing.

Currently, associative array counters and sketches are the most commonly used flow measurement architectures. However, associative array counters come at the cost of high memory usage, high computational overhead, and low operating frequency [27]. Sketches [7] are probabilistic measurement architectures requiring significantly less memory while having lower computational overhead and higher operating frequency than associative array counters. However, the advantages of sketches come at the expense of lower accuracy. A higher amount of memory has to be allocated to sketches to increase accuracy (while the counter size remains constant), resulting in a large percentage of the memory being unused, referred to as underutilization of memory. The reason for underutilization is that the larger memory size leads to fewer hash collisions. Given fewer hash collisions, the counter values will be lower due to reduced counter-sharing, leading to unused higher-order bits in many counters, thereby underutilizing memory. In addition, the amount of small or mouse flows is significant in the real network traffic data due to its skewed nature, which is also attributed to the underutilization of memory as the counter sizes are defined for large flows. Moreover, sketches are unsuitable for eviction-based algorithms, in which flows are evicted when the flow sizes fall below a certain threshold. Eviction-based algorithms require each flow to be mapped to a single counter (one-to-one mapping), making associative array counters a necessity [38].

Proposed approach: In this work, we propose a hardware-efficient sketch-based counter architecture, **SPArch** (Sketch-based Pseudo-associative array Architecture), for all-purpose flow measurement and security applications to confront the drawbacks of existing probabilistic measurement algorithms. SPArch combines the functionality of an associative array with the memory-saving benefits of sketch architectures. SPArch addresses the challenges associated with existing flow measurement solutions concerning hardware-friendliness, underutilization of memory, restricting the overall memory usage to on-chip memory, and assigning counter sizes based on flow sizes.

Furthermore, we strive to fulfill the need for a standardized measurement unit suitable for both eviction-based and sketch-based detection algorithms. As a standardized measurement unit, SPArch can serve the purpose of a standalone monitoring algorithm as well as a supporting measurement architecture. The primary focus of this work is on measurement architectures that are hardware-friendly and are capable of processing the incoming network traffic at line rate in Terabit Ethernet networks. Such architectures are particularly relevant for network security applications such as DDoS attack detection, where the flow measurement is bound to a predetermined measurement epoch.

To the best of our knowledge, none of the existing sketch-based methods employ a one-to-one mapping, akin to associative-array counters, for storing incoming flows. SPArch, on the other hand, applies one-to-one mapping to the incoming flows, allowing the flexibility to map elephant and mouse flows in separate buckets without incurring any additional memory accesses, as Section 6 explains. This functionality of mapping flows based on their sizes resembles that of some of the existing sketch architectures, such as Elastic Sketch [41] or Diamond Sketch [40], but with significantly lower memory usage. In addition to aiding the mapping of flows into appropriately sized buckets, one-to-one mapping also facilitates deletion or decrement operations. In sketches, deleting or decrementing the count value in a cell would affect the accuracy of other flows mapped to the same cell. SPArch effectively addresses this problem. Also, hash collisions increase the estimation error of sketches, whereas SPArch can detect hash collisions and identify two different flows with the same hash index as separate flows (Section 4.1).
The main contributions of this paper are:

- The proposal of a hardware-oriented measurement algorithm termed **Sketch-based Pseudo-associative array Architecture (SPArch)**, which can achieve superior accuracy with the lowest memory utilization compared to existing sketch/sketch-based measurement algorithms. SPArch can detect hash collisions. Furthermore, SPArch mitigates the underutilization of memory, a dominant drawback of sketches.
- The proposal of a flexible counter architecture that enables one-to-one mapping, allowing the organization of counters in multiple buckets without affecting the latency. The multi-bucket architecture enables the segregation of elephant flows and mouse flows into distinct buckets, facilitating the possibility of different counter sizes for mouse and elephant flows. An approximation technique is presented to scale down the sizes of the buckets to reduce the memory footprint even further.
- The evaluation of SPArch for large-flow detection.
- The implementation of SPArch on an FPGA and the evaluation of the performance.

This paper is organized as follows. Section 2 presents the problem definition and related work in network flow measurements. Section 3 introduces the preceding data structures and terms that are followed in this paper. Section 4 describes the architecture of SPArch. Section 5 provides the analysis and evaluation of the performance of SPArch and a comparison with the existing work. Section 6 explains the architecture and organization of counters in multiple buckets in SPArch. Section 7 depicts the hardware architecture of SPArch, and Section 8 provides the evaluation of SPArch on hardware and comparison with other existing architectures. Section 9 enlists the practical value of SPArch. Finally, Section 10 presents the conclusions.

2 PROBLEM DEFINITION AND RELATED WORK

This section provides a detailed description of the challenges associated with network flow measurements and the drawbacks of existing related work.

2.1 Problems in network flow measurement

Issues regarding memory requirements: In high-speed networking environments, the processing and storage of high-speed data streams require a large memory footprint. This is because high-speed data streams are characterized by a high volume of data that arrives at a very fast rate, and this data needs to be processed and stored in real-time. In order to fulfill the large storage requirements, most present-day computing systems employ off-chip Double Data Rate Dynamic Random Access Memories (DDR DRAM). However, using external DDR DRAMs considerably slows down the system, making online traffic processing nearly impossible. Moreover, DDR memory

access latency depends on various factors such as access patterns, transaction type (read or write), and the DRAM state from previous accesses. Address bus multiplexing and a bi-directional data bus also contribute to an increased access latency. The access latency of DDR3-1600 varies from 10 clock cycles (access request targeting an open row) to 72 clock cycles (access request targeting a conflicting row) for a clock with a 1.5 ns period [12]. On the other hand, the FPGA's on-chip Block RAM (BRAM) has only a single clock cycle of access latency. Constraining the memory overhead within the on-chip memory while keeping the accuracy high is the initial step to tackling the challenges associated with the implementation of measurement algorithms on hardware.

Drawbacks of existing measurement architectures: Some of the existing measurement architectures, supporting various types of detection algorithms such as intrusion detection, anomaly detection, and heavy-hitter detection, contribute well to minimizing the memory constraints using probabilistic techniques [11, 13, 19, 22, 30, 41]. However, this reduction in memory comes at the expense of reduced accuracy, higher execution delay, or complex arithmetic computations that are difficult to implement efficiently in hardware. In addition to the underutilization of memory probabilistic measurement data structures may not be suitable for all detection algorithms as different algorithms adhere to different measurement mechanisms. Large flow detection algorithms such as EARDet[38] use eviction-based measurements, whereas LOFT [30] employ sketch-based measurements.

Mapping flows of different sizes to buckets of varying sizes is an ideal solution to reduce the memory footprint while improving the relative accuracy [8]. This is because the majority of the incoming flows are mouse flows that require smaller counter sizes. Despite efforts by architectures such as Elastic Sketch [41] and count-less sketch [16] to address this problem of mapping different-sized flows to different-sized buckets, these algorithms do not tackle eviction-based measurements. This also demonstrates the reduced flexibility of probabilistic/sketch-based measurements.

When mapping a large number of network flows to corresponding buckets (one-to-one mapping) to assist eviction-based measurements, the most suitable architecture is a counter array. The prevalent approach in hardware architectures involves using associative memories to implement these counter arrays. Associative memory-based counter arrays offer support for various detection algorithms, including eviction-based algorithms, thus making them highly flexible for diverse applications. Content addressable memory (CAM) is the hardware alternative to associative memories used in most hardware platforms. However, CAM has several drawbacks compared to traditional memory technologies. It is more complex, has slower write times, operates at a lower frequency, requires a larger memory footprint, and consumes more power. Figure 1 illustrates a representation of a counter array based on a content addressable memory. Not only are there downsides to using CAM, but creating a counter array with CAM also requires additional computational logic. As shown in the figure, an address to one-hot encoder is required to generate *chip enable (CE)* signals for counters. This address to one-hot encoder is translated to a large multiplexer on hardware. The drawbacks associated with associative array/CAM memories call for the development of a more hardware-efficient alternative that offers comparable versatility.

2.2 Existing algorithmic approaches and challenges

Numerous approaches and techniques have been proposed to enhance the accuracy and efficiency of network flow measurement systems. Nonetheless, the majority of these approaches are not suitable for efficient implementation in hardware. There exist various algorithmic approaches for flow measurement architectures, such as counting algorithms, sampling-based algorithms, and sketch-based algorithms. We will provide a brief overview of these approaches, highlighting the challenges associated with translating them to hardware.

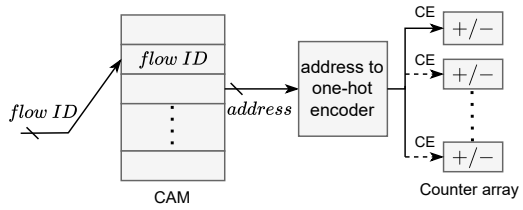

Fig. 1. CAM-based counter array

Counting algorithms process every packet in a data stream and map each flow to respective counters. These algorithms employ associative counter array-based approaches or hash table-based counters and are widely used for network measurements. Counting algorithms, such as eviction-based algorithms, prioritize storing large flows while disregarding smaller ones in order to minimize storage requirements. Exact-counting approaches on hardware that rely on associative arrays map each flow to only a single counter, calling for expensive storage architectures such as content addressable memories (CAM) [39] and additional computational overhead. Counting algorithms require storing the complete flow IDs in contrast to probabilistic data structures like sketches [7], which do not keep track of the flow IDs. This approach is unproductive unless sufficient on-chip memory is available to accommodate all the counters, which is impractical and will not scale with the growing bandwidth of networks.

There are several approaches proposed recently to improve the memory efficiency and performance of counter arrays for flow measurements. Exact-counting with shared or virtual counters and the use of hybrid SRAM-DRAM architectures somewhat reduce the memory overhead, but the extent of improvement is not substantial [4, 20]. With insufficient fast on-chip memory and an ever-increasing data rate, the feasibility of using exact counters for flow measurement is quite limited. Counter-based algorithms such as Hashpipe [33], Space-saving [22], and Lossy-counting [21] have demonstrated their efficiency in network security applications. These algorithms selectively store large flows while evicting small flows to minimize the storage requirements. However, the space-saving algorithm is ill-fitted for hardware implementations, as operations like sorting a list and finding the minimum in a list are not straightforward on hardware. Moreover, such computationally intensive operations adversely affect the performance when it comes to processing at line rate. Moreover, these algorithms suffer from significant detection errors when operating under limited memory constraints because they do not admit all the incoming flows in the flow summary. Even though the lossy-counting algorithm is more accurate than the space-saving algorithm, it is considerably slower [11]. Hashpipe guarantees accuracy but delivers a very low throughput.

Probabilistic or approximate counters [8, 14, 18, 23, 35] are one way to deal with large memory requirements, but at the cost of accuracy. Such approximated measurements and techniques such as counter sharing [4] can accommodate more data within the same memory space than exact counting techniques. However, these approaches are significantly slower [4, 15, 20], and it is challenging to deploy them at line rate. Furthermore, most of such approximation-based architectures are less hardware-efficient because of the presence of complex logarithmic computations [8, 18].

Sampling-based measurement approaches, such as NetFlow [6], sFlow [32], and ANLS [14], employ sampling in which one out of every N incoming packets are captured. The sampling rate, $\frac{1}{N}$, determines how frequently packets are captured. NetFlow employs very low sampling rates, sometimes as low as 1% or 0.01% (depending on the traffic volume), to reduce processing and storage overhead [33]. sFlow employs even lower sampling rates. While these approaches are helpful in reducing memory requirements to some extent, they come at the expense of lower accuracy due

to undersampling. This is further magnified for flow byte counting, where we count the packet size in bytes rather than the number of packets. Several recent studies have proven that high sampling causes a significant reduction in accuracy [13] and loss of information [19]. Techniques like sample and hold [9], which employ a flow table to store sampled packets, can improve the accuracy of sampling. Probabilistic data structures like hash tables can be used as flow tables, but the addition of collision-resistant mechanisms for hash tables increases the system complexity. Moreover, collision-resistant mechanisms are not easy to implement efficiently on hardware.

Sketches are probabilistic data structures that help to summarize large data streams within limited (on-chip) memory [3, 7]. Unlike sampling-based approaches, Sketch-based approaches process every packet in the incoming data stream. Sketches make use of the sub-linear space and provide a compact synopsis of the data with some loss in accuracy. Moreover, sketching algorithms can deliver better estimates than sampling/approximation-based algorithms. Nevertheless, with increasing line rates and data explosion, sketches face many challenges in keeping up with the design requirements of high-speed flow measurements. A probabilistic architecture always has a trade-off between accuracy, speed, and memory usage. Unfortunately, even sketches cannot simultaneously guarantee high accuracy and speed while operating within limited memory constraints. Sketches do not keep track of the flow identifiers, making it challenging to filter the flows based on flow sizes or delete flows at a later time. Nevertheless, because of the inherent hardware-friendliness of sketches compared to other approaches, our focus in this work primarily revolves around sketch-based methods.

2.3 A deep dive into sketch-based approaches

Sketches consist of multiple arrays of counters and are typically more memory-efficient than associative counter arrays. A count-min (CM) sketch (introduced in Sect. 3) of size (d, w) maps each flow to d counters and can simultaneously process w flows. Sketches are proven to be the most efficient measurement architectures and have been used extensively in recent times. However, there are multiple bottlenecks to take into account. The most important ones are the need for multiple independent hash computations, multiple memory accesses to process a network packet, computation-intensive arithmetic operations for counter updates, and underutilization of the allocated resources as it requires $[d \times size\ of\ the\ counter]$ bits to store a single flow.

Although sketches work on the principle of memory sharing, accuracy decreases as the number of elements exceeds w, as demonstrated in Section 5.1. Taking a real-world example for CM Sketch: for a switch having a total bandwidth of 1 Tbps and can handle 10 million flows, limiting the overestimation within 50% of the average-flow size would require approximately 250 million counters for the CM Sketch [30]. Furthermore, the amount of memory required would be much higher than mapping a flow to a single counter, which exemplifies the limitations of sketches on high-capacity routers. On hardware, large memory requirements affect not only the resource constraints but also the operating speed. Larger memory units mean more routing delays and lower operating speeds. Moreover, if the on-chip memory does not suffice, additional external memory, with a much larger access time, needs to be used.

Recently, numerous approaches based on sketches have been proposed to alleviate the complexities in network flow measurement applications, specifically in heavy-hitter detection [25, 34, 40–44]. While these approaches have shown improvements in accuracy, the use of additional data structures [25, 40, 43] or shared counters [40, 42, 44] increase the overall storage requirements. The increase in memory requirements is significant, considering the limited on-chip memory and the effect on operating speed and bandwidth requirements. One specific approach, Augmented sketch [25], employs an associated filter with a CM Sketch, and the filter stores the elephant flows

along with the key, and the corresponding flow is sent to the sketch if the filter is full. If any flow count in the sketch exceeds the smallest value in the filter, it is exchanged. Although augmented sketch provides accurate measurements for elephant flows, it does so at the expense of higher memory requirements and more memory accesses.

Elastic Sketch [41] is a well-known architecture for heavy-hitter detection, heavy change detection, and cardinality estimation. Section 3 provides a brief description of Elastic Sketch. Elastic Sketch has a *heavy part* for recording elephant flows and a light part for recording mouse flows. The light part is a simple CM Sketch. Elastic Sketch requires different hash functions for each stage of *heavy part* and the CM Sketch, which increases the computational complexity. Such a structure would keep the accuracy high, which is the primary goal of Elastic Sketch, but at the cost of increased latency and computational overhead. The hardware version of Elastic Sketch has multiple *heavy parts* operated sequentially, leading to increased latency. Moreover, the update and query operations require memory accesses equal to the sum of heavy and light parts in the worst-case scenario. The operational frequency of Elastic Sketch on hardware is also low [41], limiting the throughput. Elastic Sketch can be advantageous when the packet sizes are taken as '1' (flow size counting). However, it becomes less suitable for flow byte counting as the packet sizes can be up to 64KB if we consider jumbo frames, making the 8-bit counter size in the light part of Elastic Sketch inadequate.

SF sketch [42] also uses distinct sub-sketches for mouse and elephant flows, necessitating additional memory to enhance accuracy, which is undesirable. Diamond Sketch [40] and One Memory Access sketch [44] use multiple levels of smaller sketches. As the lower level counters overflow for an incoming flow, that flow is moved to the next higher level. Diamond Sketch incorporates additional data structures like *deletion* and *carry* parts, and both Diamond Sketch and One Memory Access Sketch involve the storage of the flow fingerprints for improved accuracy. Consequently, the memory demand is high for both of these approaches. Furthermore, the worst-case number of memory accesses is proportional to the number of levels, adversely affecting the speed and making pipelining difficult on hardware. Bloom Sketch [43] and SA Sketch [45] introduce a combination of Bloom filter and sketch-based approach, and both employ multiple layers of sketches. Such multi-layer architectures would adversely affect the worst-case memory access time, operating speed, and memory utilization.

In addition to sketches, there exist data structures that are based on sketch architectures and achieve higher accuracy than sketches. One such example is HeavyKeeper, which is briefly described in Section 3. HeavyKeeper algorithm is used for measuring top-k elephant flows and incorporates fingerprint along with the count in each bucket to improve the accuracy. HeavyKeeper assumes minimal fingerprint collisions, and to minimize fingerprint collisions, the sizes of the fingerprints should not be too small. Storing a large fingerprint along with the count causes a significant increase in memory usage. Despite HeavyKeeper not storing all flows like the CM Sketch, the inclusion of a fingerprint makes the memory usage more or less comparable. Moreover, if the fingerprints do not match, HeavyKeeper applies exponential decay, introducing the possibility that an incoming large flow may collide with an existing large flow. Another drawback with sketch/sketch-based algorithms, such as HeavyKeeper and Elastic Sketch, is that they are not designed for flow byte counting but for flow size counting.

Recently introduced algorithms such as LOFT [30] and ALBUS [29] take memory reduction as their primary design criteria while achieving a high detection accuracy against specific attacks like large flow and pulsating attacks. Both these algorithms employ probabilistic measurement approaches while accepting hash collisions. These techniques reduce the overall memory requirement. However, the additional logic employed to compensate for the hash collisions is not completely hardware-friendly, even though the performance is excellent on software. The introduction of

Table 1. Comparison of counting architectures

Type of data structure	Measurement error	Memory overhead	Latency on hardware (in *ns*)	One-to-one mapping	Deletion support	Computational complexity	Hardware friendliness
Exact counting	**No error**	Very high	Low to Very high	**Yes**	**Possible**	Moderate to High	Low
Sampling-based	Very high	Moderate to high	Moderate to Very high	**Possible**	**Possible**	High to Very high	Very low
Sketch-based	Low to moderate	**Low**	Very low to moderate	Not possible	Not possible	**Low to Moderate**	Moderate to High
SPArch	Very Low	**Low**	**Very low**	**Possible**	**possible**	**Low to Moderate**	**Very high**

these algorithms also puts forward the interesting scenario of combining multiple algorithms that target distinct attacks to detect various types of attacks. In a scenario where multiple individual algorithms are combined to tackle a range of attack patterns, it would be wise to share resources, such as flow measurement and storage architectures, to minimize resource utilization on hardware. This leads to the requirement of a standard measurement module that can handle both associative array-based (or eviction-based) and probabilistic flow measurements.

2.4 Synopsis of the related work

An overview of different approaches along with our proposed approach, SPArch, is presented in Table 1. Based on the analysis of the related work, it can be inferred that sketches face the challenge of striking a poor trade-off between memory usage and accuracy, which is concerning, although they are better than the other probabilistic counting techniques. Moreover, they lack the versatility of individual flow mapping provided by architectures like associative array-based counters. Associative array-based counters facilitate one-to-one mapping, allowing us to predict the counter address to which incoming flows will be mapped. This gives the flexibility to organize the counters into multiple buckets (of different counter sizes if required). In real networks, the majority of the flows are mouse flows, and elephant flows are rare [1]. Hence, keeping equally-sized counters for both elephant and mouse flows is not wise. An efficient way to handle this is by allocating separate counter arrays to store them. Some recent works exploit this technique [41, 42], but such data structures demand more memory overhead, partially negating the benefit of having separate buckets.

Knowing that only a small percentage of flows would require a large counter, a one-to-one mapping architecture could be space-saving as it is easier to map the flows into appropriately sized buckets. This also gives the freedom to choose different scaling factors for different counters/buckets if we need to apply approximation, and this could bring in a massive reduction in memory overhead. Moreover, the challenges in hardware implementations are manifold. Even though sketches achieve higher operating speeds than associative array-based counters, they still require considerable memory to ensure higher accuracy. This also leads to lower operating frequencies as the routing delays increase with an increasing number of memory blocks.

3 PRELIMINARIES

Counting the total size of packets in a network flow is termed network flow measurement. All network packets that possess the same flow identifier (ID) form a network flow. the flow ID is extracted from the packet header and is usually defined by the 5-tuple ⟨source IP address, source port, destination IP address, destination port, protocol ID⟩. The flow ID can also be represented based on the different levels of granularity/detail, such as the source IP address, a combination of source IP address and source port, any combination of the 5-tuple, or the 5-tuple itself. The finer the granularity, the more bits are required to represent the flow ID. This work uses a combination of source and destination addresses and ports to form the flow ID. The protocol field from the 5-tuple is omitted to obtain a flow ID of size 96 bits.

The size of the network flow can be measured either in terms of the number of packets in a flow or the byte volume. This paper defines 'flow size' as the number of packets and 'flow bytes' as the byte volume. A flow is categorized as an *elephant* flow if the total volume over a measurement period surpasses a given high-bandwidth threshold. When the total volume over a measurement period falls below the low-bandwidth threshold, we refer to the flow as a *mouse* flow. An incoming flow is mapped as a pair of flow ID x and size s, as (x, s).

To analyze the efficiency of SPArch, it is essential to compare the performance of SPArch with state-of-the-art architectures. While choosing the algorithms for comparison, the Count-Min sketch (CM Sketch) serves as the baseline. SPArch also follows the basic data structure of CM Sketch. Currently, Elastic Sketch is one of the most efficient algorithms for applications like heavy-hitter detection, outperforming nearly all state-of-the-art detection algorithms, including UnivMon [19], FlowRadar [17], Hashpipe [33], and Reversible Sketch [31]. Elastic Sketch is compared against all these algorithms and proved to be better. Another highly efficient algorithm that uses fingerprints similar to SPArch is HeavyKeeper [11]. HeavyKeeper proves to be more efficient than other algorithms, such as Space-saving [22] and Lossy counting [21]. Therefore, a comparison with Elastic Sketch and HeavyKeeper seemed to be sufficient to demonstrate the effectiveness of SPArch. We implemented both these algorithms and compared them against SPArch.

A brief overview of Count Min Sketch, HeavyKeeper, and Elastic Sketch is provided here. The two primary operations of these architectures are the update operation and the query operation. The former updates the architecture with an (x, s) pair, while the latter performs a lookup of s, given a flow ID x.

Count Min sketch: The data structure of Count Min (CM) sketch is shown in Fig. 2. A CM Sketch is a 2-dimensional array, having width w and depth d, of counters. A CM Sketch is well-suited for hardware because of the ease of parallel implementation and low memory utilization.

Update An incoming flow ID x_i is mapped to multiple counters in the array using d pairwise independent hash functions h_j, where j=1, .., d. Each counter indexed by one of the d hash functions is updated as: $counter[j, h_j] \leftarrow counter[j, h_j] + inc_i$, where h_j represents the hash value of x_i by the j_{th} hash function. The value with which these counters are incremented represents the incremental size of flow x_i, where $inc_i = s$, with s either the flow size or flow bytes.

Query While querying, the incoming flow ID x_i is hashed d times, choosing d counters. The minimum counter value amongst all the hash-indexed counters is taken: $estimate = min(counter[j, h_j], \forall j \in 1..d)$.

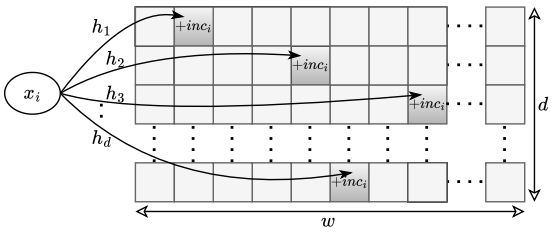

Fig. 2. Data structure of CM Sketch

HeavyKeeper: The data structure of HeavyKeeper is shown in Fig. 3. It is a 2-dimensional array comprising of d rows, where each row consists of w buckets. Each bucket has two fields: a fingerprint and a counter. Each incoming flow x_i is mapped to d buckets using d independent hash functions h_j, where j=1,...,d. Another hash function h_f is used to generate the fingerprint from the flow ID.

Each bucket is represented as $A_j[h_j(x_i)]$, and the fingerprint (FP) and counter fields in the bucket are represented with $.FP$ and $.C$ suffixes, respectively.

Update The update operation is based on three different strategies. If $A_j[h_j(x_i)].C = 0$, the fingerprint field is set as $A_j[h_j(x_i)].FP = f_i$ and the counter field is set as $A_j[h_j(x_i)].C = 1$, where f_i is the fingerprint of flow x_i. If $A_j[h_j(x_i)].C > 0$ and $A_j[h_j(x_i)].FP = f_i$, the counter field is updated as $A_j[h_j(x_i)].C = A_j[h_j(x_i)].C + inc_i$, where inc_i is the incremental size of flow x_i, where $inc_i = s$. When $A_j[h_j(x_i)].C > 0$ and $A_j[h_j(x_i)].FP \neq f_i$, the proposed count-with exponential-decay strategy is applied to this bucket and $A_j[h_j(x_i)].C$ is decayed with a probability.

Query While querying, the incoming flow ID x_i is hashed d times, choosing d buckets. The maximum counter value of all hash-indexed buckets that have a matching fingerprint is taken: $estimate = max(A_j[h_j(x_i)].C, \forall j \in 1..d \iff A_j[h_j(x_i)].FP = f_i)$.

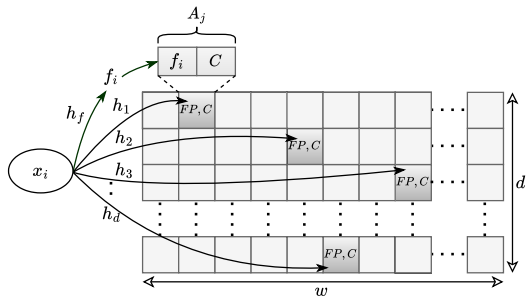

Fig. 3. Data structure of HeavyKeeper

Elastic Sketch: The data structure of the basic version of Elastic Sketch is shown in Fig. 4. Elastic Sketch consists of a *heavy part* for storing elephant flows and a *light part* for storing mouse flows. The *heavy part* is a simple hash table, where each bucket in the hash table stores the key (flow ID), positive votes ($v+$), negative votes ($v-$), and a $flag$. The number of packets belonging to the flow present in the bucket is indicated by $v+$, while $v-$ stores the number of packets belonging to any other flow mapped to the same bucket. The incoming flows are mapped to the *heavy part* using the address generated by a hash function h. The *light part* is a CM Sketch. An incoming flow is first mapped to the *heavy part*. Based on the update scenarios, the flow is then evicted to the *light part* or vice versa if required.

Update During the update, the incoming flow x_i is mapped to the *heavy part* first using the hash value $h(x_i)$. If the mapped bucket is empty, $(x_i, 1, 0, F)$ is inserted into that bucket. $flag = F$ indicates that no eviction has happened in the bucket. If the mapped bucket is not empty and $x_i = key$, the value of $v+$ is incremented by 1. If the mapped bucket is not empty and $x_i \neq key$, the value of $v-$ is incremented by 1. If $\frac{v-}{v+} \geq \lambda$, then the stored flow is evicted and $(key, v+)$ is inserted into the *light part*, and the *heavy part* is stored with $(x_i, 1, 1, T)$. Here, λ is a predefined threshold, and $flag = T$ indicates the eviction in the bucket. If $x_i \neq key$ and $\frac{v-}{v+} < \lambda$, $(x_i, 1)$ it is inserted into the CM Sketch.

Query During the query operation, the size is queried from the CM Sketch if the flow is not present in the *heavy part*. If the flow is present in the *heavy part* and $flag = F$, the size is the value of $v+$. If the flow is present in the *heavy part* and $flag = T$, then the values are retrieved from both heavy and *light parts*, and their sum gives the actual size.

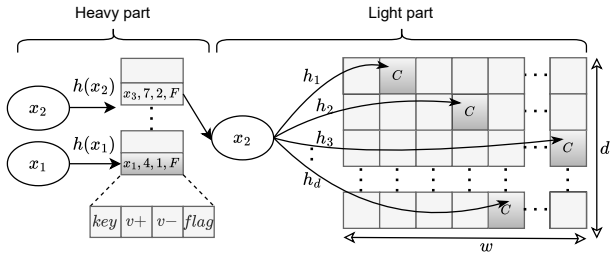

Fig. 4. Data structure of Elastic Sketch

4 SKETCH-BASED PSEUDO-ASSOCIATIVE ARRAY ARCHITECTURE (SPARCH)

The data structure we propose in this paper, SPArch, is a 2-dimensional array of size (d, w) with an associated 1-dimensional counter memory array of size (N_c), as shown in Fig. 5. The depth of the sketch is d, also representing the number of hash functions required to map each flow to the sketch. The width of the sketch is w, representing the number of cells/buckets in each row. Each cell in the sketch, termed fingerprint-address cell or FAC, comprises two fields: a fingerprint field f and an address field A. An address generator, a simple $log_2 N_c$-bit up-counter, is used to generate the address A of the counters, and the generated address is stored in an address register (*Address Reg* as shown in the figure). The update and query operations of SPArch are discussed in detail in the following sections.

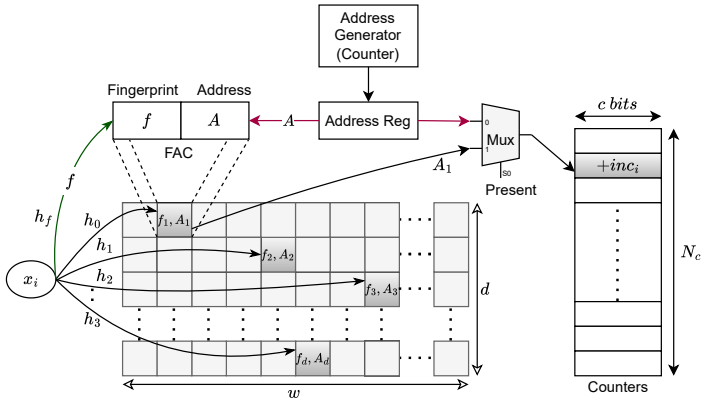

Fig. 5. Data structure of SPArch

4.1 Update operation

An incoming flow x_i is mapped to the FAC using d independent hash functions h_j, where j=1,...,d. Each FAC is represented as $FAC_j[h_j(x_i)]$ and consists of the fingerprint-address pair (f, A). A unique fingerprint is generated by hashing the flow ID using a hash function h_f. The size of the fingerprint is determined to achieve optimal accuracy while keeping it to a minimum possible value. The size of the fingerprint (f) in bits is denoted by q. The fingerprint must be non-zero, and if a generated fingerprint happens to be zero, it is assigned the maximum possible value of $2^q - 1$. If any of the FACs have a fingerprint value of zero, it indicates that the corresponding element is not present. Conversely, if all fingerprint values are greater than zero, the presence of the element depends on whether there is a matching fingerprint. Instead of using separate hash functions to

Table 2. Notational conventions of SPArch

x	Incoming flow
inc_i	Incremental flow size
d	Depth of the sketch/number of hash functions
w	Width of the sketch
N_c	Number of counters
n	Number of distinct flows
A	Address of the counter
FAC	Fingerprint-Address Cell (A Bucket)
$FAC_j[h_j(x)]$	FAC in the jth row having the address $h_j(x)$
f	Fingerprint
h_j	Hash function to index the jth row of the sketch
h_f	Hash function to generate fingerprint
q	Size of the fingerprint in bits
c	Size of the counter in bits
a	Address size of counters in bits ($log_2 N_c$)

generate the sketch indices and fingerprint, a single hash function is employed. This hash function generates the required hash bits, which are then divided into d hash indices and the fingerprint. The total number of hash bits required is $d \times log_2 w + q$. Xoodoo-NC [26] is the hash function used to generate the required hash values, which proved to be fast and exhibits excellent avalanche properties.

The counter address A in the FAC points to the counter memory to be updated with the incremental flow size inc_i. When a new incoming flow is updated, the address of the counter A is retrieved from the address register (*Address Reg* as shown in Fig. 5) and written to the FAC along with the fingerprint, if the FAC is not yet occupied. Once the update is complete, a new address is generated by an address generator. This newly generated address is then stored in the address register for the next incoming flow update. The counter memory array has N_c counters, each with a size of c bits. SPArch can handle n flows simultaneously, where n=N_c, and n can even be greater than w, whereas the other sketches, such as CM Sketch, can only handle w flows simultaneously. Choosing a value of w much lower than n still yields high accuracy for SPArch, and Section 5.1 provides a detailed analysis. The size of the address part A is a bits, where $a = log_2(N_c)$. For a comprehensive list of the notational conventions used in the SPArch architecture, please refer to Table 2.

There can be multiple scenarios in the update operation, and the subsequent sections discuss all of these scenarios. Algorithm 1 describes the complete algorithm of the update operation. Implementing multiple scenarios for update and query operations helps to enhance the hardware-oriented design of SPArch. Irrespective of the depth of the sketch and the number of counters, there is a memory access latency of only three clock cycles for the update operation (sketch read, counter read, and sketch and counter write) and two clock cycles for query operation (sketch read, counter read). Each row in the sketch can be accessed in parallel as each row is a separate memory block. Additionally, the sketch and the counter write operations can be executed together within a single clock cycle. All the update and query computations can be performed in parallel once the memory read operation is complete. Consequently, the multiple update/query scenarios do not introduce any extra latency, resulting in a time complexity of $O(1)$.

4.1.1 If the element x_i is not present.

Case 1: Some (or all) of the locations indexed by the hashes are empty This scenario is visualized by the sub-figures (a) and (b) in Fig. 6. In this scenario, a new address A_x is obtained

from the address generator. The empty FACs in the hash-indexed locations of the sketch will be updated with the fingerprint f_x and the address A_x: $FAC_j[h_j(x)] \leftarrow (f_x, A_x)$.

After sampling, the value of the address generator is incremented. The non-empty locations will not be overwritten. Finally, the counter memory location is updated with the newly added address A_x and with the corresponding incremental size *inc* (lines 10-14, Algorithm 1).

Algorithm 1: Update operation

1 Parameters: Width of sketch - w, Depth of sketch - d, Number of counters - N_c, Size of the counter (bits) - c, Incoming flow at time t - x_t, Fingerprint of x_t - f_{x_t}, Flow size of x_t - s_t
2 **Initialize():**
3 Create a $d \times w$ sketch array and a $N_c \times 1$ counter array. All initialized to 0.
4 Address generator value initialized to 0 ($AdGen[value] = 0$).
5 $inc = s$
6 **Update**(x, inc):
7 $f_x = h_f[x], A_x = AdGen[value]$
8 **for** $j \leftarrow 1$ *to* d **do**
9 $(f_j, A_j) \leftarrow FAC_j[h_j[x]]$
10 **if** $f_j = 0, \exists\ j$ **then**
11 $FAC_j[h_j[x]]_{f_j=0} \leftarrow (f_x, A_x)$
12 $Counter[A_x] = Counter[A_x] + inc$
13 $AdGen[value] = AdGen[value] + 1$
14 **end**
15 **else if** $f_j \neq 0, \ \forall\ j$ **then**
16 **if** $f_j \neq f_x, \ \forall\ j$ **then**
17 $\forall\ FAC_i[h_j[x]]_{min(A_j)} \leftarrow (f_x, A_x)$
18 $Counter[A_x] = Counter[A_x] + inc$
19 $AdGen[value] = AdGen[value] + 1$
20 **end**
21 **else if** *($f_j \neq f_x$ and $FAC_j[h_j[x]]$is same, $\forall\ j$)* **then**
22 $FAC_j[h_j[x]]_{j=1)} \leftarrow (f_x, A_x)$
23 $Counter[A_x] = Counter[A_x] + inc$
24 $AdGen[value] = AdGen[value] + 1$
25 **end**
26 **else if** $f_j = f_x \ \exists\ j$ *and* A_j *is same* $\forall\ f_j = f_x$ **then**
27 $Counter[A_j] = Counter[A_j] + inc$
28 **end**
29 **else if** $f_j = f_x \ \exists\ j$ *and* A_j *is not same* $\exists\ f_j = f_x$ **then**
30 **if** *Any* A_j *has majority* **then**
31 $A_j = majority(A_j)|_{f_j=f_x}$
32 $Counter[A_j] = Counter[A_j] + inc$
33 **end**
34 **else**
35 $A_j = max(A_j)|_{f_j=f_x}$
36 $Counter[A_j] = Counter[A_j] + inc$
37 **end**
38 **end**
39 **end**
40 **end**

Case 2: None of the hash-indexed locations are empty, none of the fingerprints match, and fingerprint-address pairs in all locations are different This case is represented by the sub-figure (c) in Fig. 6. In this scenario, the fingerprint-address pairs having the lowest address (which represents the oldest flow inserted) in the FAC is replaced with the incoming fingerprint and the address pair: $FAC_j[h_j(x)]_{min(A)} \leftarrow (f_x, A_x)$.

After sampling, the value of the address generator is incremented. Finally, the counter memory location at A_x is initialized with the incremental size *inc* (lines 16-20, Algorithm 1). The address of the evicted flow is not reused until the next reset period.

To ensure that the highest address values are indeed the newest, SPArch denies any new updates until the new measurement period in case of an overflow, where none of the N_c counters are empty, and the address generator has reached its maximum counter value. There may theoretically arise a security vulnerability as an adversary can send arbitrary amounts of traffic without being recorded once the address generator reaches its maximum value and SPArch denies updates. However, this vulnerability cannot practically happen if we ensure that the number of counters N_c, which is the maximum value generated by the counter, chosen must be enough to accommodate all possible flows for the measurement period. Moreover, double insurance can be guaranteed by adding an extra bit to the address generator counter, which doubles the number of addresses generated, ensuring that the address generator does not reach its maximum value before the measurement period.

Case 3: None of the hash-indexed locations is empty, none of the fingerprints matches, and ALL/SOME locations have an identical fingerprint-address pair This scenario is represented by the sub-figure (d) in Fig. 6. In this scenario, the identical fingerprint-address pair in the first row is replaced with the incoming fingerprint-address pair: $FAC_j[h_j(x)]_{j=1} \leftarrow (f_x, A_x)$. The value of the address generator is incremented. The counter memory location at the newly added address is updated with incremental size *inc* (lines 21-24, Algorithm 1).

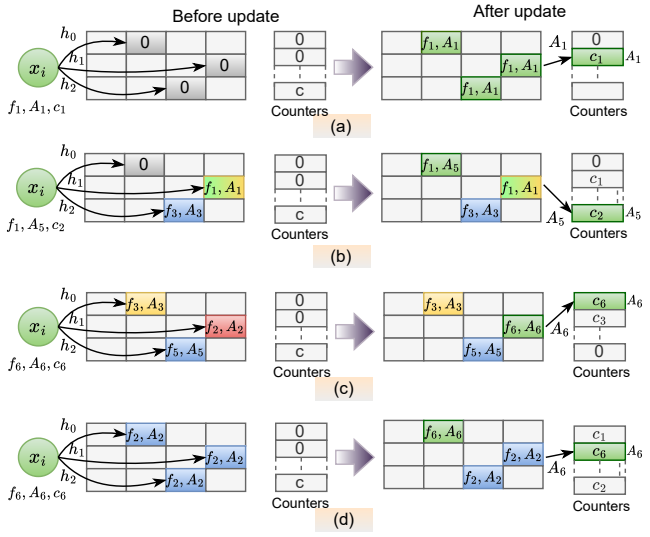

Fig. 6. Update scenarios if the flow is not present

4.1.2 *If the element x_i is present.*

Case 4: One or more of the hash-indexed locations have a unique matching fingerprint, and all these matching fingerprints are paired with the same addresses This scenario is represented by sub-figures (a) and (b) in Fig. 7. In this case, the counter memory is updated with the address from the matching fingerprint-address pair with incremental size *inc* (lines 26-28, Algorithm 1).

Case 5: Two or more of the hash-indexed locations have a matching fingerprint, these matching fingerprints are paired with multiple different addresses This scenario is represented by sub-figures (c) and (d) in Fig. 7. In this case, the address which occurs the most is used to update the counter memory. If no address has a majority, the address of the most recent fingerprint-address pair is used (Lines 29-38, Algorithm 1).

4.1.3 *Collision resistance.*

SPArch exhibits excellent collision resistance and is able to detect hash collisions during update operations. The collisions can be hash-index collisions, fingerprint collisions, or both. Cases 2 and 3 of the update operation represent the occurrence of hash-index collisions where the incoming flow is mapped to already existing locations. Cases 4 and 5 of the update operation represent the occurrence of fingerprint collisions as well as the occurrence of both fingerprint and hash-index collisions. In the case of a hash-index collision, the fingerprint is used to identify the collision and distinguish the element. In case of a fingerprint collision or both hash-index and fingerprint collision, the stored address acts as a unique identifier to distinguish the elements and detect the collision.

Nevertheless, there may have very rare scenarios where collisions cannot be detected. For example, a flow x with fingerprint f is mapped to four FACs, (f, A), (f, A), (f, A), and $(none, none)$. As per Case 1, the last FAC will be updated as (f, B). A recurrence of x would result in updating the counter A as per Case 5 based on the majority, as there is a fingerprint collision. However, such scenarios are extremely rare that occur due to hash collisions. The hash function Xoodoo-NC ensures that the hash output is completely random for each flow ID. Xoodoo-NC maps a 96-bit input to a 96-bit output space and has near-perfect avalanche scores, ensuring sufficient randomness to limit the simultaneous occurrence of fingerprint and hash-index collisions for the scenario described above to happen. Considering the birthday attack, an average collision occurs only after encountering around $1.17 * 2^x$ flows, where $x = min(d * lg(w) + k, 96)/2$.

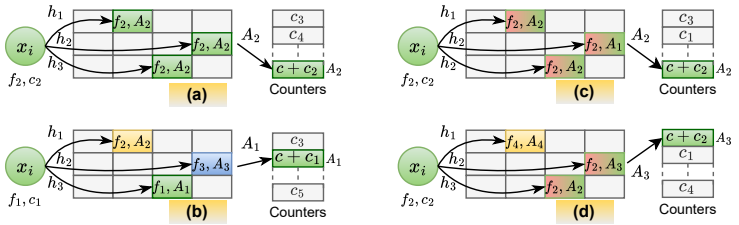

Fig. 7. Update scenarios if the flow is present

4.2 Query operation

The query operation also has to consider multiple scenarios. Algorithm 2 provides the complete algorithm of the update operation.

Case 1: Some (or all) of the locations indexed by the hashes are empty

Case 2: None of the locations indexed by the hashes have a matching fingerprint Sub-figure

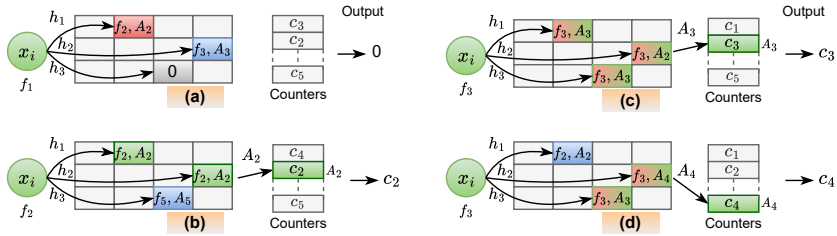

Fig. 8. Query scenarios

(a) in Fig. 8 represents these two scenarios. In both cases 1 and 2, the counter value is zero (lines 5-7, Algorithm 2).

Case 3: None of the locations indexed by the hashes are empty; one or more fingerprints match; and all the corresponding addresses in the fingerprint-address pair are identical This scenario is represented by the sub-figure (b) in Fig. 8. The counter value can be obtained through the unique address from a matching fingerprint-address pair (lines 8-10, Algorithm 2).

Case 4: None of the locations indexed by the hashes are empty; one or more fingerprints match; but the corresponding addresses are different This scenario is represented by sub-figures (c) and (d) in Fig. 8. The counter value at the most frequently occurring address is retrieved. If no address has a majority, the fingerprint-address pair having the most recent address is selected and used to retrieve the count (lines 11-20, Algorithm 2).

Algorithm 2: Query operation

```
1  Estimate(x):
2  f_x = h_f[x]
3  for j ← 1 to d do
4      (f_j, A_j) ← FAC_j[h_j[x]]
5      if (f_j = 0, ∃ j) OR (f_j ≠ 0 and f_j ≠ f_x, ∀ j) then
6          Estimate = 0
7      end
8      else if (f_j ≠ 0 ∀ j) and (f_j = f_x ∃ j and A_j is same ∀ f_j = f_x) then
9          Estimate = Counter[A_j]
10     end
11     else if (f_j ≠ 0 ∀ j) and (f_j = f_x ∃ j and A_j is not same ∃ f_j = f_x) then
12         if Any A_j has majority then
13             A_j = majority(A_j)|_{f_j=f_x}
14             Estimate = Counter[A_j]
15         end
16         else
17             A_j = max(A_j)|_{f_j=f_x}
18             Estimate = Counter[A_j]
19         end
20     end
21 end
```

4.3 Deletion operation

The deletion operation follows the same process as the update operation. If the element to be deleted is not present in the sketch, the deletion operation is halted. If the element is present, the

FAC(s) that contain(s) the matching fingerprint-address pair is/are emptied, and the corresponding counter is reset.

4.4 Preliminary theoretical analysis

In this section, we provide a preliminary theoretical analysis of SPArch, considering all *update* scenarios. We examine the probability that the $(n+1)^{th}$ flow is correctly recorded and the expected number of FACs that record the $(n+1)^{th}$ flow.

4.4.1 Probability of Correct Recording. After n flows, the probability that the $(n+1)$-th flow is correctly recorded is analyzed by considering two cases:

(1) All FACs collide, but no fingerprint collisions occur.
(2) Some FACs are empty.

Thus, the probability that the $(n+1)^{th}$ flow is correctly recorded is given by:

$$\Pr\left[(n+1)^{th} \text{ flow is correctly recorded}\right] \geq \Pr[\text{Case 1} + \text{Case 2}] \approx p^d e^{-\hat{p}d} + (1 - p^d)$$

Here, let:

$$p = \Pr\left[(n+1)^{th} \text{ flow has FAC collision(s) in a row}\right]$$

$$\hat{p} = \Pr[\text{FP also collides, given FAC collision}]$$

The probability p can be expressed as:

$$p = 1 - \Pr\left[(n+1)^{th} \text{ flow does not collide with the previous } n \text{ flows in any FAC in a row}\right]$$

$$= 1 - \left(1 - \frac{1}{w}\right)^n \approx 1 - e^{-\frac{n}{w}}$$

(Note: the approximation $1 + x \approx e^x$ assumes x is small, based on the expansion of Taylor series.)

The probability $\hat{p}$ is given by:

$$\hat{p} = \frac{1}{2^q}$$

***Case 1**: All FACs Collide, but No Fingerprint Collisions.*

$$\Pr[\text{All FACs collide}] = \prod_i \Pr[\text{collision in row } i] = p^d$$

$$\Pr[\text{No FP collisions}] = (1 - \hat{p})^d \approx e^{-\hat{p}d}$$

(Note: as above, the approximation assumes $\hat{p}$ is small.)

Therefore:

$$\Pr[\text{All FACs collide, but no FP collisions}] \approx p^d e^{-\hat{p}d}$$

***Case 2**: Some FACs are Empty.*

$$\Pr[\text{At least one FAC is empty}] = 1 - \Pr[\text{All FACs collide}] \approx 1 - p^d$$

4.4.2 Expected Number of FACs Updated. After n flows, the expected number of FACs that record the $(n+1)^{th}$ flow is given by:

$$E[\text{Number of FACs updated}] = d \cdot E[\text{FAC empty in row } i] + 1 \cdot \Pr[\text{All FACs collide, but no FP collisions}]$$

$$\approx d \cdot (1 - p) + p^d e^{-\hat{p}d}$$

5 EMPIRICAL ANALYSIS OF SPARCH

This section provides an in-depth empirical analysis of SPArch as a standalone monitoring algorithm. It should also be noted that the analysis holds true for the functionality of SPArch as a supporting associative array architecture, working in tandem with the other algorithms. The simulation-based analysis is performed with varying parameter sizes. A comparison is drawn between SPArch and other related algorithms, namely CM Sketch, HeavyKeeper, and Elastic Sketch. These algorithms are also implemented with the corresponding parameter sizes to ensure a fair comparison. The analysis focuses on two key aspects: flow measurement and large flow detection. A threshold is set for the latter, and the detection results are analyzed. Since SPArch is designed as a counter array for network flow measurement tasks, it lacks the inherent ability to perform tasks such as heavy-change detection, entropy estimation, and cardinality estimation on its own. To perform these tasks, SPArch needs to be integrated into dedicated algorithms. Therefore, the analysis focuses solely on flow measurement and the detection of large flows. The dataset used for the analysis is a subset of the CAIDA traffic trace from 2018, measured on a 10 Gbps link, modified for detecting large flows [2]. The CIC-DDoS2019 [10] dataset from the Canadian Institute of Cybersecurity (CIC) is also used for verification of network flow measurement. The results of the CAIDA dataset are marked as solid lines and of CIC dataset are marked as dotted lines in the charts. As the results from the CIC dataset closely mirror those from CAIDA dataset, the following sections will only provide explanations based on the CAIDA dataset.

The analysis is carried out for the following metrics:

Accuracy: Accuracy defines the rate of correct estimation of the flow size compared to the actual size. Accuracy = $\frac{\hat{n}}{n}$, where $\hat{n}$ is the number of correctly estimated flows and n is the total number of flows inserted.

Average Absolute Error (AAE): AAE defines the mean absolute error rate of the estimated flow size with respect to the actual flow size. AAE = $\frac{1}{|X|}\Sigma_{x_i \in X}|\hat{n_i} - n_i|$, where X is the set of all flows x_i, and $\hat{n_i}$ and n_i are the estimated and real flow sizes of flow x_i respectively.

Average Relative Error (ARE): ARE defines the mean relative error rate of the estimated flow size with respect to the actual flow size. ARE = $\frac{1}{|X|}\Sigma_{x_i \in X}\frac{|\hat{n_i} - n_i|}{n_i}$, where X is the set of all flows x_i, and $\hat{n_i}$ and n_i are the estimated and real flow sizes of flow x_i respectively.

Precision: Precision refers to the fraction of correctly identified large flows present in the set of all large flows detected by the sketch. Precision = True Positives (TP) / (True Positives (TP) + False Positives (FP)).

Recall: Recall refers to the fraction of flows correctly detected as large flows by the sketch out of all the large flows that exist in the link. Recall = True Positives (TP) / (True Positives (TP) + False Negatives (FN)).

It is to be noted that the number of counters, N_c, in SPArch is always equal to the number of distinct flows to be handled, n. Hence, n also represents the total number of counters N_c in SPArch.

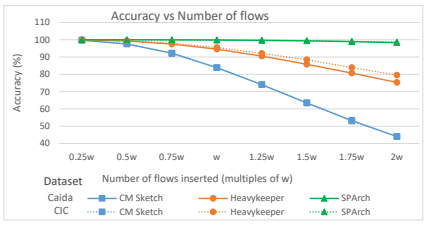

Fig. 9. Accuracy vs number of flows

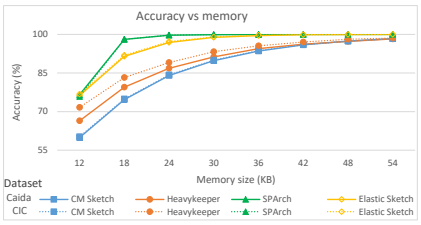

Fig. 10. Accuracy vs memory size

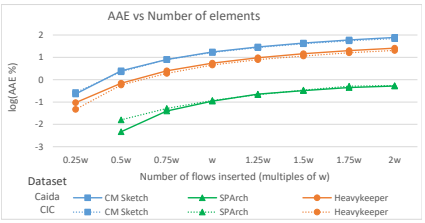

Fig. 11. AAE vs number of flows

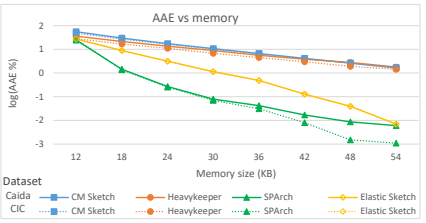

Fig. 12. AAE vs memory size

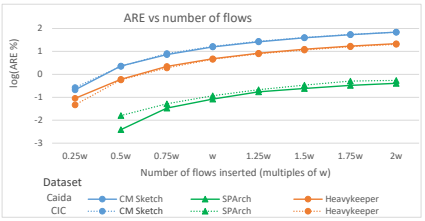

Fig. 13. ARE vs number of flows

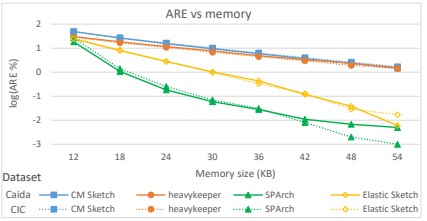

Fig. 14. ARE vs memory size

5.1 Evaluation of Accuracy

The accuracy is evaluated against the number of flows inserted and memory size. To evaluate against the number of flows n inserted, all algorithms maintain a constant width w of 1024 and depth d of 4. The purpose of evaluating against the number of flows inserted is to demonstrate SPArch's ability to handle a high influx of incoming flows. For evaluating against the memory size, the values of w and n are taken as 2048 and the value is d is taken as 4 for each algorithm. The fingerprint (FP) size of HeavyKeeper is kept as 12 bits. The authors of HeavyKeeper [11] set the fingerprint size as 16 bits for $d = 2$. As per our analysis, a fingerprint size of 12 bits is sufficient enough to provide the same accuracy as having a fingerprint size of 16 bits when $d = 4$, making HeavyKeeper more memory-efficient. The fingerprint size of SPArch is set to 8 bits for evaluation against the number of flows inserted. Conversely, for evaluating against the memory size, the fingerprint size and depth d are varied accordingly to achieve the desired memory size. Since the data structure of Elastic Sketch has substantial differences from other algorithms, the analysis is solely performed based on the memory size. The number of buckets in the *heavy part* is taken as 10% of the total number of elements n to be stored, and the *light part* has a width w equal to 90% of n. For Elastic Sketch, the counter size of *key* equals the flow ID size, and the sizes of $v+$ and $v-$ are 24 bits. The counter sizes in the *light part* are 12 bits, assuming that no jumbo frames are considered.

Accuracy versus the number of flows *n:* Fig. 9 depicts the accuracy with respect to the number of flows inserted as a multiple of w. The decrease in accuracy as the number of elements increases is minimal for SPArch, and its accuracy remains close to 100%. Even with $n{=}2 \times w$, SPArch achieves an accuracy of 98.3%, whereas it is 75.3% and 44.0%, respectively, for HeavyKeeper and CM Sketch. For a counter size of 24 bits, CM Sketch and SPArch, SPArch with a fingerprint size of 8 bits, utilize a memory size of 12KB. In contrast, HeavyKeeper requires 18KB to accommodate the same number of flows.

Accuracy versus memory size: Fig. 10 shows the plot of accuracy against the memory size. It is evident from the plot that SPArch achieves notably higher accuracy compared to other algorithms with identical memory constraints. SPArch achieves ~99% accuracy with only 18KB of memory for

n=w, whereas CM Sketch and HeavyKeeper require >300% more memory to achieve a similar level of accuracy. At a memory size of 12KB, Elastic Sketch and SPArch exhibit similar accuracy, but Elastic Sketch requires 200% more memory than SPArch to achieve 99% accuracy. The *heavy part* of Elastic Sketch is a hash table, and the hash table experiences an average collision rate of $\frac{n.(n+1)}{2w}$. Hence, for $n = w$, the average number of collisions in the *heavy part* is $\sim\frac{n}{2}$. The scenario where the size of the hash table w=0.1n results in a large number of collisions, which in turn gives rise to most of the incoming flows being mapped to the *light part*, even if the *heavy part* is not even 20% filled. Apparently, this causes the accuracy of the Elastic Sketch to be equal to the accuracy of CM Sketch. Notably, SPArch uses full-size counters, while 90% of the counters in Elastic Sketch are half-sized, making the memory reduction achieved by SPArch highly significant. With a larger counter size, SPArch would achieve even greater memory savings compared to the other algorithms.

5.2 Evaluation of measurement error: average absolute error (AAE) and average relative error (ARE)

The AAE is evaluated against the number of flows inserted and the memory size. For evaluating against the number of flows inserted, the width w is set to 1024 for all algorithms, and the depth of the sketch d is kept as 4. For evaluating against memory size, the values of w and n are taken as 2048, and the value is d is taken as 4 for each algorithm to ensure a fair comparison. The counter size is 24 bits. The fingerprint (FP) size of HeavyKeeper is 12 bits to provide the best memory-accuracy trade-off. The fingerprint size of SPArch is kept as 8 bits for evaluation against the number of flows inserted. The fingerprint size and depth d of SPArch are varied accordingly for the required memory size while evaluating against memory size. For Elastic Sketch, the analysis is performed only based on memory size. The parameters for Elastic Sketch are chosen as described in Section 5.1.

5.2.1 *Average Absolute Error (AAE).*

AAE versus the number of flows inserted: Fig. 11 shows the plot of $log_{10}(AAE)$ against the number of flows inserted. The plot shows significant differences between SPArch and the other algorithms. When $n = w$, the values of AAE are ~50× and ~159× higher for HeavyKeeper and CM Sketch, respectively, compared to SPArch. As n approaches $2 \times w$, the error becomes ~78% for CM Sketch and ~26% for HeavyKeeper, whereas the AAE is only 0.52% for SPArch, showcasing its exceptional inherent accuracy.

AAE versus memory size: Fig. 12 illustrates the plot of $log_{10}(AAE)$ against memory size. For a memory size as small as 12KB, the AAE difference of SPArch was only 2.2× and 1.5×, respectively, compared to CM Sketch and HeavyKeeper. As the memory size increases from 12KB to 18KB, the difference becomes a significant 21× and 15×, respectively, compared to CM Sketch and HeavyKeeper. As the memory size becomes as high as 54KB, the AAE for SPArch reduces by ~4070× to reach 0.006%. At this point, the difference with CM Sketch and HeavyKeeper becomes ~271× and ~298×, respectively. It is very much evident from the analysis that SPArch possesses high accuracy, and the absolute error is considerably reduced even with a relatively smaller increment in memory.

5.2.2 *Average Relative Error (ARE).*

ARE versus the number of flows inserted: Figure 13 illustrates the relationship between the number of inserted flows and the plot of ARE. Both HeavyKeeper and CM Sketch exhibit a gradual increase in ARE as the number of flows increases. However, HeavyKeeper achieves an ARE ~3-4× lower than CM Sketch, for instance, 4.8% for $n = w$ and 21.8% for $n = 2w$ compared to 15.6% and 69.1% of CM Sketch. SPArch maintains a remarkably low error rate of only 0.4% even when $n = 2w$, making it 173 times lower than CM Sketch and 39 times lower than HeavyKeeper. The

characteristics of Elastic Sketch fall between SPArch and the other two algorithms. Elastic Sketch offers better AAE than CM Sketch and HeavyKeeper, and reaches nearly zero when the memory size is 54KB or more. However, Elastic Sketch has a significantly higher error rate than SPArch. Furthermore, it is worth noting that 90% of the counters in Elastic Sketch use only 50% of the counter sizes than the other algorithms, and with 100% counter sizes, the error rate of Elastic Sketch will closely follow the AAE of CM Sketch.

ARE versus memory size: The plot of ARE against the number of flows inserted is shown in Fig. 14. As seen from the figure, the curves of CM Sketch and HeavyKeeper closely align with each other, in contrast to SPArch, as observed in the plot of AAE. HeavyKeeper and CM Sketch exhibit nearly identical error rates with an equivalent memory allocation, indicating that HeavyKeeper demands a larger memory capacity in order to achieve higher accuracy. While the error rate is high for SPArch when the memory is 12KB, it drops below 1% as the memory size surpasses 18KB, eventually approaching zero. For a memory size of 54KB, the ARE of SPArch is ~300× better than both of the other algorithms. The ARE of Elastic Sketch demonstrates similar characteristics as observed in the AAE plot and falls between SPArch and the other two algorithms. The ARE of SPArch rapidly diminishes to zero, whereas the Elastic Sketch achieves an ARE of zero when the memory size reaches 54KB or more.

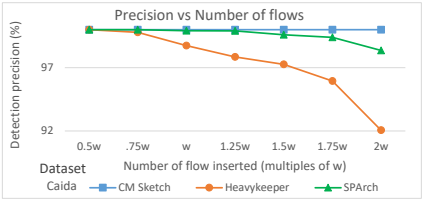

Fig. 15. Flow detection: precision vs number of flows



Fig. 16. Flow detection: precision vs memory size

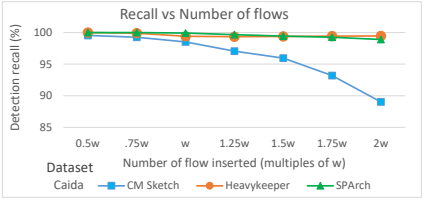

Fig. 17. Flow detection: recall vs number of flows

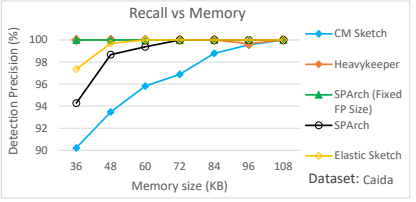

Fig. 18. Flow detection: recall vs memory size

5.3 Evaluation of large flow detection: precision and recall

To analyze the detection of large flows where flow sizes exceed a predefined threshold, the values of w and d are set as 16384 and 4, respectively, for all the algorithms. The duration of the measurement cycle is set as 100ms. The fingerprint sizes of SPArch and HeavyKeeper are 8 bits and 12 bits, respectively. Similarly, the analysis of Elastic Sketch is performed solely on the basis of memory size. A threshold is set, and the flows whose aggregated sizes exceed the threshold are classified as large flows. The precision and recall are then evaluated against the number of flows inserted and the memory size.

Precision and recall versus the number of flows inserted: Fig. 15 and Fig. 17 display the plots of precision and recall, respectively, against the number of flows. As depicted in Fig. 15, CM Sketch maintains a precision of 100% for any value of n, indicating that CM Sketch identifies all large flows without any misses. However, upon examining the plot of recall, the values gradually decrease as n increases. When $n = w$, the recall of CM Sketch is 98.5% and declines to 89% when $n = 2 \times w$. This is alarming as CM Sketch can falsely classify a benign flow as malicious, leading to false positives. The characteristics of HeavyKeeper are somewhat opposing to CM Sketch. The precision of HeavyKeeper experiences a significant decrease with the increase in w. For HeavyKeeper, the precision is 98.7% when $n = w$, which falls to 92% when n becomes $2 \times w$. It indicates that HeavyKeeper does not always filter the large flows out. Nevertheless, HeavyKeeper exhibits satisfactory results for recall, consistently achieving a recall rate greater than 99% even when $n = w$.

SPArch performs equally well on both precision and recall characteristics. SPArch holds a 100% precision and recall when $n = w$. The deterioration of precision is negligible as n increases. For instance, SPArch maintains the precision to 99.4% when $n = 1.75 \times w$ and 98.4% when $n = 3 \times w$. The decline of recall is also negligible and is always greater than 99%. Compared to CM Sketch, which achieves 100% precision due to overestimation, SPArch is on par when $n = w$, and there is only a slight decrease in precision as n increases. The precision of HeavyKeeper is the lowest as it underestimates the flows. On the other hand, the overestimation of CM Sketch causes the misclassification of benign flows as malicious, resulting in blocking the legitimate flows. SPArch maintains a high recall percentage, indicating minimal false classifications. It is very much evident from the analysis that SPArch offers greater accuracy than the other algorithms. Although the precision of SPArch slightly diminishes for higher values of n when $n > w$, it does not imply underestimation like HeavyKeeper. The decrease in precision is mainly due to the replacement of the older flows with newer flows, as hash collisions become significant (Section 4.1, case-2 and case-3), which happens only when the value of n significantly exceeds w.

Precision and recall versus memory size: Precision and recall versus the memory size are plotted in figures 16 and 18, respectively, providing insights into the performance of the algorithms under different memory conditions. For evaluation against memory, $n = 4096$ is kept constant for each algorithm. In the case of HeavyKeeper, the fingerprint size is fixed at 12 bits, while the value of w is adjusted accordingly to meet the memory requirements. The size of the *heavy part* of Elastic Sketch is taken as $0.1n$, and the size of the *light part* is varied accordingly to fulfill the memory requirements. For SPArch, two cases are considered to meet the memory requirements. The first case involves a fixed FP size (set as 12-bits) while varying size of d (referred to as "SPArch (Fixed FP size)" in the figures). The second case involves varying both FP size ($4 \leq$ FP size ≤ 9) and d (referred to as "SPArch" in the figures). The latter case has a value of d one row higher than SPArch with a fixed FP size.

CM Sketch consistently maintains a precision of 100%, whereas recall becomes the lowest when the memory size is small due to significant overestimation. With increased memory size, the accuracy and recall of CM Sketch improve. HeavyKeeper exhibits a recall of 100% and reaches a precision of 100% when the memory size surpasses 60KB. Elastic Sketch is tailored for detecting elephant flows, exhibiting a 100% precision and near-hundred percent recall for a memory size of 36KB or more. As the overestimation property of Elastic Sketch closely aligns with CM Sketch, its precision remains consistently high. SPArch, on the other hand, offers the flexibility to tailor the results based on specific requirements. SPArch with varying FP size attains a precision of 100%, while SPArch with fixed FP size achieves a recall of 100%. Although the precision of SPArch with fixed FP size is low for a memory size of 36KB, it reaches 100% for memory sizes above 48 KB. Similarly, the recall for SPArch with varying FP size is close to 99% when the memory size is greater

than 48KB and reaches 100% at a memory size of 72KB. Hence, compared to other algorithms, SPArch is more flexible in applying manipulations based on the memory requirements to achieve a 100% precision or recall.

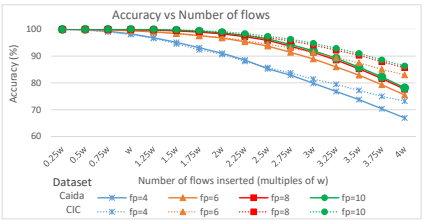

Fig. 19. Accuracy vs number of flows (SPArch)

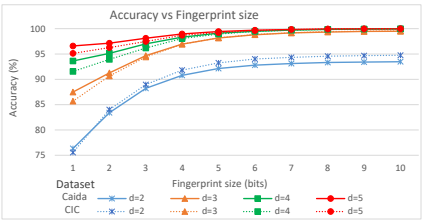

Fig. 20. Accuracy vs Fingerprint size (SPArch)

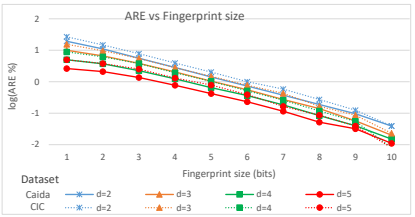

Fig. 21. ARE vs Fingerprint size (SPArch)

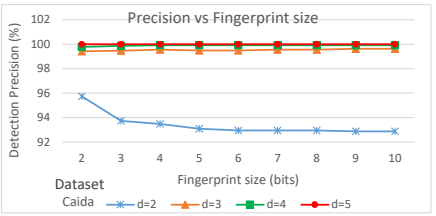

Fig. 22. Flow detection: precision vs FP size (SPArch)

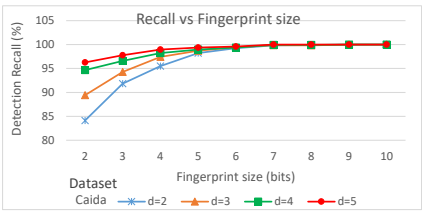

Fig. 23. Flow detection: recall vs FP size (SPArch)

5.4 Evaluating the performance of SPArch

To support the accuracy statements of SPArch, the analysis of accuracy, measurement error, precision, and recall of SPArch against varying fingerprint size q and depth d are presented here. This analysis provides insights into the high accuracy achieved by SPArch, even when employing smaller fingerprint sizes and lower depths.

5.4.1 Accuracy analysis of SPArch with varying fingerprint size and depth d.

Fig. 19 plots the accuracy of SPArch against the number of flows inserted, for various fingerprint sizes and a fixed depth of $d = 4$. To assess the stability of SPArch's accuracy, the number of flows inserted is increased to $4 \times w$. With a fingerprint size equal to 4, SPArch achieves an accuracy of

98.3% for $n = w$ =1024 and 91.0% for $n = 2 \times w$, surpassing both CM Sketch and HeavyKeeper while requiring less memory. With a fingerprint size greater than or equal to 6, SPArch can maintain near-hundred percent accuracy for $n = 2 \times w$, and keep the accuracy higher than 75% for $n = 4 \times w$.

Fig. 20 illustrates the relationship between the accuracy of SPArch and the fingerprint sizes for $w = n$ = 1024 and different values of d. Even with a fingerprint size of 2 bits and $d = 4$ (total memory requirement of 9KB), SPArch can achieve an accuracy exceeding 95%. This accuracy surpasses that of HeavyKeeper while utilizing only 50% of the memory required by HeavyKeeper. Additionally, when $d = 2$ and a FP size of 2 bits, SPArch attains an accuracy of 83.4% (total memory requirement of 6KB). This accuracy outperforms CM Sketch while utilizing only 50% of the memory compared to CM Sketch. Consequently, SPArch can achieve near-hundred percent accuracy with the lowest memory requirement.

5.4.2 Measurement error of SPArch with varying fingerprint size and depth d.

The analysis of ARE on SPArch is performed with varying depth and fingerprint sizes, and the results are plotted in Fig. 21. Remarkably, even with $d = 2$ and a fingerprint size of 5 (corresponding to a memory size of 6.75KB and $n = w$ =1024), SPArch surpasses the best attainable ARE values of CM Sketch and HeavyKeeper, where both CM Sketch and HeavyKeeper are allocated a memory size of 54KB, as indicated in Figure 14. The error rates decrease gradually with increasing depth or fingerprint size, as illustrated in the figure. This characteristic is particularly advantageous in hardware implementations, allowing us to limit the usage of block RAMs by carefully choosing the number of rows of the sketch d or the fingerprint size to achieve a desired level of accuracy.

5.4.3 Precision and recall of SPArch versus fingerprint size and depth d.

Figures 22 and 23 present the plot of precision and recall, respectively, for varying fingerprint sizes of SPArch. When the FP size is at its lowest, the precision for $d = 2$ is slightly higher due to a significant overestimation error. However, it stabilizes at 93% when the FP size exceeds 3. When $d \geq 3$, the precision is ∼100% even when the FP size is as small as 2. As for the recall, it is lower for smaller FP sizes and gradually increases with increasing FP size. When the FP size exceeds 5, the recall attains ∼100% as the accuracy becomes maximum. This analysis underlines the earlier statement that SPArch offers greater flexibility, allowing users to select the memory size based on their preference for higher precision, higher recall, or both, considering the constraints of available memory.

6 OPTIMIZING SPARCH: ORGANIZING COUNTERS IN MULTIPLE BUCKETS

Only a small proportion of the incoming flows would be classified as heavy-hitters, necessitating the use of large counters. Sketches, in general, allocate the same counter sizes for all the flows, resulting in underutilization of the allocated memory. Allocating different counter sizes for elephant and mouse flows is impossible for sketches as sketches do not follow one-to-one mapping like hash tables or associative arrays. However, SPArch offers the benefit of a flexible counter organization due to its one-to-one mapping. One way to leverage this property is by organizing flows into distinct buckets with different counter sizes, allocating smaller counter sizes to mouse flows and larger counter sizes to elephant flows. Keeping separate buckets for elephant flows eliminates the possibility of removal of an elephant flow during collisions where the oldest address is removed. As the addresses allocated to elephant flows will be the largest, they cannot be removed during collisions when a mouse flow is already present in the counters. Another advantage is that approximation can be applied to different buckets with different scaling factors. Since such an implementation is advantageous on hardware, a hardware-oriented design is described here. The counter buckets can be organized in two ways: each bucket as an independent memory block or multiple buckets within the same memory block. The implementation details are discussed in the subsequent sections.

6.1 Counter buckets: as separate memory blocks

Fig. 24 illustrates one approach to organizing counters into separate buckets. The counters are arranged in two buckets, b_m and b_e, with b_m dedicated to storing mouse flows and b_e designated for elephant flows. The total number of counters is N_c, with 80% of them allocated to b_m and the remaining 20% allocated to b_e. The number of bits allocated to each counter in b_m and b_e is c_m and c_e, respectively. Unlike SPArch, where a counter serves as the address generator, a FIFO, initialized with the addresses, is used as the address generator for bucket b_m, while b_e employs a counter as its address generator. It is possible to have a total of N buckets where the size of each subsequent bucket is larger than the preceding buckets. For all the buckets except the last one, FIFOs are employed as the address generators, whereas a counter is used as the address generator for the final bucket.

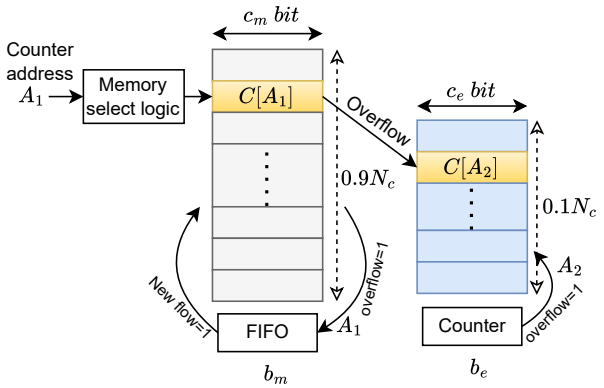

Fig. 24. Organization of buckets: separate memory blocks

Update and Query operations: During the update process, the sketch sends the address (A_1) of the counter to be updated to bucket b_m. The address is read from the FIFO if the incoming flow is new. The counter corresponding to A_1 is updated, and if the counter overflows, it is moved to the address A_2 in the bucket b_e. The address of the counter A_2 is generated using the counter. The address A_1 of the overflown counter is written back to the FIFO, which will only be re-used once the limit of the address range has been reached. The new address A_2 is written to the FAC, replacing the existing address. Since we split the total number of counters into two buckets, one for mouse and one for elephant flows, an issue may arise. What if there are no elephant flows, and the counters for mouse flows are entirely occupied, while there are new incoming flows to be processed? In this case, if all the mouse counters are occupied, and the FIFO becomes empty, the next incoming flow is automatically stored in the elephant counters. Hence, as long as the number of flows to be stored is less than or equal to the total number of counters N_c, no flows will be missed. The query operation remains the same as discussed in Section 4.

Since each bucket is a separate memory block, the addresses of these blocks must be unique. Hence, a memory-select logic is employed to differentiate and map the addresses to the buckets. To keep the memory-select logic simple, an additional bit is introduced at the most significant bit (MSB) of the addresses. This creates a simple multiplexer serving as the memory-select logic. The value of the extra bit is '0' for the first bucket, and the value is '1' for the second bucket. While writing the address to the sketch, the corresponding bit at the MSB is appended. Typically, the existing MSB is used as the appending bit unless an overflow occurs. In case of an overflow, the

MSB is incremented and appended with the new address. This keeps the computation requirements to a bare minimum. For the number of buckets (N) larger than 2, the number of extra bits to be added is $log_2 N$.

Latency: In contrast to alternative methods that employ multiple memory blocks to segregate flows, our approach maintains consistent latency. Transitions between buckets occur only when there is an overflow. During an overflow, the second bucket is not read but only written. By writing both the sketch and counter memory simultaneously, it ensures that latency remains unaffected, regardless of whether there is an overflow.

Approximate counting: Applying approximation to counters can reduce memory usage when precise measurement values are not required. For finding heavy-hitters that exceed a certain threshold, accurate measurements are not necessary. However, considering that mouse flows are relatively small, it would be unwise to apply the same scaling factors to buckets containing both mouse and elephant flows, as this could result in significant accuracy discrepancies for mouse flows. Therefore, having dedicated buckets for elephant flows simplifies the process of approximating the counts. In the recent study titled "A-CM Sketch" by Sateesan et al. [28], a hardware-oriented approach called HSAC is introduced, which requires only 16 bits to represent a 41-bit counter.

6.2 Counter buckets: in the same memory block

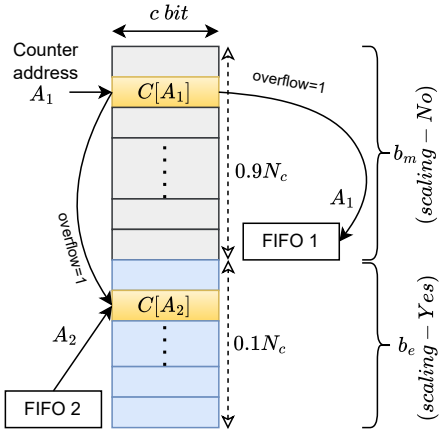

Fig. 25. Organization of buckets: single memory block

Another way of organizing counters involves keeping multiple buckets in the same memory block, which helps in eliminating the memory-select logic and additional MSB bit to differentiate the counter buckets. Figure 25 illustrates the organization of these buckets. The memory block has a depth of N_c, which is divided into two (or more) imaginary sections. The first $0.9N_c$ counters are designated for storing mouse flows, while the remaining $0.1N_c$ counters are used to store elephant flows. In contrast to the counter buckets using separate memory blocks, FIFOs are used as the address generators for all buckets. When employing a single memory block for all the buckets, the size of all the counters must be the same, adhering to the width of the memory block. Hence, approximation or scaling is necessary to increase a bucket's storage capacity. Using HSAC [28] approximation, only 16 bits are required to count up to 2^{41}. As a result, both b_m and b_e can be set to 16 bits, with approximation or scaling enabled in b_e. If more than two buckets are present,

the scaling factor can be adjusted for each bucket, allowing for varying levels of approximation according to the flow sizes.

Update and Query operations: When the counter having an address A_1 placed in bucket b_m overflows during the update operation of a preexisting flow, it is moved to bucket b_e, simultaneously reading the new address A_2 from FIFO2. The new address of the counter is written to the FAC, replacing the existing address. The original counter address A_1 is written back to FIFO1. Whenever a new flow arrives, it is typically written to b_m using the address obtained from FIFO1 unless all counters in b_m are already occupied. If all the counters in b_m are occupied, the next incoming flow is mapped to b_e. The query operation remains unchanged, as discussed in Section 4, except that the estimated value is computed based on the approximate count for the values stored in b_e.

Approximating the count while moving from b_m to b_e: Approximating or scaling the measurements while moving the values from b_m to b_e is accomplished using HSAC. Initially, the count is rescaled to fit the format of HSAC. Also, a minor modification is applied to HSAC, such that we divided the counter into only two sections consisting of the exponent part (exp) and the counter part ($count$). The counter size of c-bits of bucket b_e is divided into an l-bit exponent part and a k-bit counter part. The algorithm 3 presents the pseudo-code of the approximation process. The function $approximate(V_1)$ is applied explicitly during the transition from b_m to b_e, while the $update(count, exp, inc)$ function is executed for all counter update operations in b_e.

Latency: The latency is similar to the counter buckets with separate memory blocks. The counter memory is read only once and written only once, making no increment in latency.

Algorithm 3: Approximation operation

Parameters:V_1=Actual count, V_2=Approximated count, exp=l-bits, $count$=k-bits
`approximate(`V_1`)`:
  Initialize:
    $count = \frac{2^k}{2}$
    $exp = c - k - 1$
    $inc = V_1 - 2^k$
  end
  update($count, exp, inc$)
end
`update(`$count, exp, inc$`)`:
  $count = count + \lfloor \frac{inc}{2^{exp}} \rfloor$
  With probability $\{\frac{inc}{2^{exp}}\}$: $count = count + 1$
  if $count > 2^k$ **then**
    $exp = exp + 1$
    $t_1 = count$
    $count = \frac{count}{2}$
    With probability $\{\frac{t_1}{2}\}$: $count = count + 1$
  end
end
Return$\{V_2 = [exp, count]\}$

6.3 Memory reduction

In addition to the benefit of having separate buckets to classify flows, there is a further reduction in memory requirements for the counters. Assuming 90% of the counters ($0.9N_c$) are allocated to the mouse flows, and the counter size is reduced by half for mouse flows, the total counter

memory requirement is reduced by 45% without approximation. While applying approximation to elephant flows with the size of the counters reduced by half, the total counter memory requirement is reduced by 50%. However, there is no reduction to the sketch memory size as the fingerprint and address sizes remain unaltered. Assuming a fixed fingerprint size of 8 bits and $n = N_c = w$, the overall reduction in memory for $w = 65536$, $d = 4$ is 11.25% without approximation and 12.5% with approximation.

6.4 Effects of multi-bucket optimization and the reduction in complexity of multi-bucket counters

Multi-bucket counters significantly reduce the memory overhead in SPArch and employing FIFOs instead of counters as the address generators reduces some computational requirements. However, incorporating FIFOs can add some complexity to the datapath and requires initialization time during startup. Furthermore, the overflown counter address written back to the FIFOs can cause a reduction in accuracy as the old addresses will be reused at the end. Nevertheless, the accuracy variation will be insignificant because only the last $0.1N_c$ elements will be using the reused addresses. Also, the removal of the re-used overflown addresses in the worst case of a collision (Case 2 of update operation) will not cause the eviction of any elephant flows as the elephant flows are in a separate bucket. These issues can be resolved by allocating a total of $1.1N_c$ counters: N_c counters for mouse flow bucket b_m and an additional $0.1N_c$ counters for the elephant flow bucket b_e. With this approach, reusing the addresses of b_m is no longer necessary, and a simple counter that serves as an address generator can replace the FIFO. The *empty* signal of the FIFO can be replaced by the *full* signal of b_m when the count value of the address generator reaches N_c. Similarly, the FIFO for b_e can be replaced by another counter having an initial value of N_c. This way, the complexity can be kept to a minimum by increasing the overall memory requirement of the multi-bucket counters by only $0.1N_c \times c_m$, where c_m represents the counter size of mouse counters. The problem of re-used addresses can also be avoided by denying entries once the memory is full. As the size of the memory is determined based on the measurement period, it is highly unlikely that the memory will be exhausted. In addition, while using approximate counters can impact accuracy, the loss of accuracy is negligible when counter sizes are greater than 10 bits [28], as they do in this context.

7 HARDWARE ARCHITECTURE

Fig. 5 shows a high-level view of the SPArch data structure, which comprises the sketch and the counter array. The sketch component is represented by a two-dimensional array with dimensions of $w \times d$, and the counter array consists of N_c counters. Fig. 26 depicts the hardware architecture diagram of the SPArch data structure.

The implementation of the sketch and counter array utilizes true-dual-port Block RAMs (BRAM). The control unit is the brain of the architecture, which controls the sequence of operations to be performed. From the network packet header, a 96-bit flow ID is extracted and the hash values are computed. The hash-computation module is a crucial component of the hardware architecture and significantly impacts the latency of probabilistic data structures. A non-cryptographic hash function Xoodoo-NC [26] is employed to hash the flow ID. Xoodoo-NC is fast, has low logical depth, and exhibits excellent avalanche properties. The output of Xoodoo-NC can be any multiple of 96 bits. The resulting hash output is split into d hash values of size $log_2 w$ bits and a fingerprint of size q-bits. The hash values map the flow ID to the FACs, while the fingerprint helps determine the presence or absence of the incoming flow.

SPArch stands out from other sketch architectures primarily due to its unique counter array of N_c counters, and notably, N_c can be greater than the width of the sketch w. Unlike other sketches, where exceeding the number of counters by more than w can result in decreased accuracy (Section 5.4),

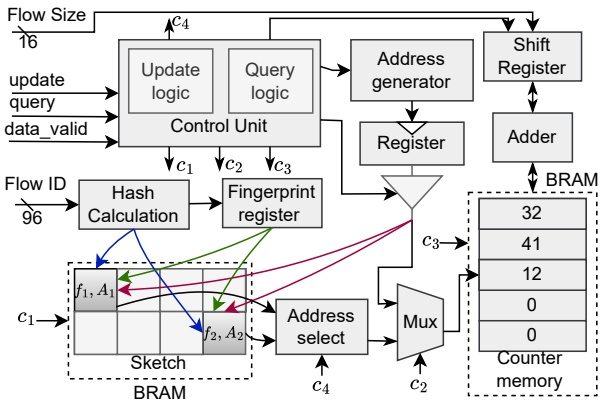

Fig. 26. Hardware architecture of SPArch

SPArch offers greater flexibility in terms of counter quantity. For assigning flows to counters, an address generator generates addresses that are used to map each flow to a specific counter. The address-select logic determines the address of the counter to be updated/queried if an incoming flow is present in the sketch. The shift register acts as a delay element to compensate for the lookup time of the sketch before the flow size computations can be performed.

The latency of the hash function, Xoodoo-NC, is a single clock cycle. The hash-indexed sketch memory is read in the subsequent clock cycle. The sketch is implemented with separate memory blocks for each row, allowing parallel execution of memory read and write operations within a single clock cycle. The update logic, in conjunction with the address-select logic, determines the address of the counter to be updated, and the counter memory is updated accordingly. The read/write operations of the counter memory also require only one clock cycle. The sketch and counter memory write operations are performed simultaneously in a single cycle, keeping the number of extra cycles required for SPArch compared to CM Sketch to only one. Even in the worst-case scenario, where a new packet needs to be processed in every clock cycle, SPArch can process it efficiently through pipelining. The parallel read/write capabilities of all the memory blocks facilitate pipelining easier. The true-dual-port BRAM ensures seamless execution of simultaneous read and write operations within a single cycle on a memory block when needed. Moreover, even during pipelining, there is no need to perform two simultaneous read or write operations within a single cycle for the sketch or counter memory blocks, even in the worst-case scenario.

8 HARDWARE EVALUATION

8.1 Experimental setup

The algorithm is evaluated on an FPGA using the Vivado 2020.2 design tool, specifically targeting the Alveo U250 Data Center Accelerator Card (xcu250-figd2104-2L-e). The parameters are chosen to allow for a fair comparison with other algorithms. The memory organization of CM Sketch, HeavyKeeper, and SPArch is kept similar for a fair comparison. The original hardware version of Elastic Sketch consists of several sub-tables of the *heavy part* connected sequentially, leading to a significant increase in memory accesses and latency. Hence, the simplified and hardware-friendly version of Elastic Sketch with a single *heavy part* is implemented, where the number of buckets in the *heavy part* of Elastic Sketch is 12.5% of the total number of counters. In order to minimize the hardware complexity and improve the throughput of Elastic Sketch, the division operations are converted to right-shift operations by approximating the denominators to the power of two.

This approximation is applied only in the hardware implementation to achieve a better hardware performance, and the accuracy is computed as given in the analysis provided in Section 5. Only the sketch part of HeavyKeeper is implemented, and the fingerprint size is fixed to 12 bits, as mentioned in the original literature. The pseudo-random number generation for HeavyKeeper is implemented using a linear feedback shift register (LFSR). As the number of counters of SPArch is equal to the number of distinct flows, the parameter n is used to represent both.

8.2 Accuracy versus resource utilization

To ensure a fair comparison, the performance of CM Sketch, HeavyKeeper, Elastic Sketch, and SPArch was assessed primarily based on accuracy. Since the accuracy with respect to memory usage varies significantly for these algorithms, evaluating these algorithms based on any other parameter would not be equitable. Additionally, comparing them solely on the basis of the depth of the sketch d would also be unfair, as the relative increase in the memory requirement of SPArch with an increase in the value of d is much lower than other algorithms. It is also evident from Section 5 that the error, precision, and recall of flow measurements are proportional to the accuracy. Although SPArch has a significant advantage in terms of relative memory consumption as d increases (d is also the depth of the *light part* for Elastic Sketch), the evaluation results are presented only for a constant value of d=4. The comparison provided in Table 3 is performed based on accuracy as the main parameter. Although the optimized version of Elastic Sketch employs d=1, it deteriorates the accuracy compared to using d=4 with the same memory allocation. Hence we chose d=4 to achieve the best memory-accuracy trade-off. The original Elastic Sketch paper also indicates that d=3 or 4 would be required to achieve high accuracy. Our empirical evaluation of Elastic Sketch supports this, showing that with a 54KB memory allocation, d=4 achieves an accuracy of 99.8%, whereas d=1 only reaches 94.2%. To achieve accuracy comparable to d=4, more memory would be needed if d=1. Moreover, the entire memory allocated translates to a single large block of memory for d=1, compared to four smaller blocks of memory for (d=4). A large memory block might also increase the routing delays in hardware implementation. Another possible advantage of keeping d=1 is that the number of independent hash functions can be changed from four to one. However, we already use a single hash function, Xoodoo-NC, to generate all hash values. So, there are no additional hash computation requirements irrespective of the value of d. The evaluation is performed for n=16384. The fingerprint size of SPArch is kept at 8 bits. The counters, including the *heavy part* of Elastic Sketch, are sized as 32 bits. The counter size of the *light part* of Elastic Sketch is taken as 16 bits, as the incoming flow size can be up to 16 bits.

Table 3 provides the details regarding the size of the data structures, the total theoretical memory requirement, and the hardware evaluation results for various configurations (n=16384 for all configurations) of all the implemented algorithms, aimed at achieving a specific level of accuracy. As previously mentioned in Section 5, the memory requirement to achieve ~100% accuracy is the lowest for SPArch. To achieve an accuracy of at least 99.8%, SPArch requires only 54 BRAMs, whereas CM Sketch, HeavyKeeper, and Elastic Sketch require 4.3×, 5.9×, and 2.4× more BRAMs, respectively. These results prove that reducing the underutilization of memory drastically reduces the memory requirements. The memory usage of SPArch can be further reduced by reducing the fingerprint size without any drastic changes in accuracy, as demonstrated in Section 5.1, and by organizing counters in multiple buckets and applying approximation, as discussed in Section 6.

Although HeavyKeeper demonstrates superior software efficiency compared to CM Sketch, its architecture is not optimized for hardware, resulting in poor performance in terms of latency and memory utilization. Additionally, the relative increase in memory (BRAM) utilization with respect to accuracy is higher for HeavyKeeper than for the other algorithms. Elastic Sketch, while also experiencing a relatively high increase in memory utilization concerning accuracy, still performs

Table 3. Comparison of hardware resources and latency of SPArch against CM Sketch and HeavyKeeper

	Accuracy	Sketch size	Memory requirement	LUT	BRAM	Operating Frequency	Update Latency	Throughput
CM Sketch	84.04	w=16384,d=4	2048 Kbit	**690**	58	270 MHz	**14.8 ns**	138 Gbps
	99.02	w=43008,d=4	5376 Kbit	**1059**	154	240 MHz	**16.6 ns**	123 Gbps
	99.77	w=65536,d=4	8192 Kbit	**1041**	232	216 MHz	**18.5 ns**	111 Gbps
HeavyKeeper	84.12	w=11392,d=4, FP size=12bit	1958 Kbit	1090	58	247 MHz	16.2 ns	126 Gbps
	99.02	w=32768,d=4, FP size=12bit	5632 Kbit	1132	160	220 MHz	18.2 ns	112 Gbps
	99.90	w=65536,d=4, FP size=12bit	11264 Kbit	1375	320	188 MHz	21.2 ns	97 Gbps
Elastic Sketch	84.07	LP:w=14464,d=4; HP:w=2048	1226 Kbit	1471	39	196 MHz	35.7 ns	100 Gbps
	99.01	LP:w=38400,d=4; HP:w=2048	2722 Kbit	1651	83	186 MHz	37.7 ns	95 Gbps
	99.84	LP:w=65536,d=4; HP:w=2048	4418 Kbit	1654	129	185 MHz	37.7 ns	95 Gbps
SPArch	84.03	w=4096,d=4, FP size=8bit	**864 Kbit**	1424	**24**	**294 MHz**	17.0 ns	**151 Gbps**
	99.00	w=16384,d=4, FP size=5bit	**1728 Kbit**	1504	**48**	**260 MHz**	19.25 ns	**133 Gbps**
	99.93	w=16384,d=4, FP size=8bit	**1920 Kbit**	1522	**54**	**258 MHz**	19.4 ns	**132 Gbps**

FP-fingerprint, HP-heavy part, LP-light part

better than HeavyKeeper in this regard. However, the number of LUTs required for Elastic Sketch is higher than that of other algorithms because of the hardware unfriendliness causing additional hardware logic requirements. It should be noted that the hardware resource requirement of Elastic Sketch would be considerably higher if the approximated division optimization had not been performed in the hardware implementation. SPArch, on the other hand, proves to be more efficient on hardware, and the relative increase in memory usage with respect to accuracy is the lowest among the others. As the accuracy varies from ∼99.0% to at least 99.8%, the increase in memory usage of HeavyKeeper is ∼100%, while CM Sketch and Elastic Sketch experience a range of 50% to 60% increases. In contrast, SPArch only sees a modest increase of around 12%. From this analysis, it is evident that SPArch exhibits a remarkable combination of high accuracy and significantly low memory requirements, rendering it an optimal architectural choice.

8.3 Latency and throughput

Both HeavyKeeper and CM Sketch have the same latency since the operations involved are similar, such as hashing, memory read/write, and counter update. For SPArch, one additional counter memory read/write operation is required. Nevertheless, only one extra cycle, which is counter memory read, is required for SPArch because the sketch memory write operation can be performed simultaneously with the counter memory read/write operation. This leads to a latency that is only one clock cycle more than CM Sketch and HeavyKeeper. Despite the extra cycle latency, SPArch has a higher operating frequency than Elastic Sketch and HeavyKeeper for the same memory overhead. Additionally, SPArch operates at a higher frequency compared to the other algorithms while achieving similar levels of accuracy. SPArch employs separate memory blocks for each row of the sketch and the counter array. By using multiple smaller BRAMs instead of a single large block of memory, SPArch achieves better placement and routing, reducing the routing delay [28]. Since all memory blocks can be accessed in parallel, pipelining becomes more effortless, enabling SPArch to process a packet in every clock cycle. The basic version of Elastic Sketch with a single heavy and light part, however, has a worst-case latency of seven clock cycles.

Assuming a minimum packet size of 64 bytes and a worst-case scenario where a packet is received in every cycle, SPArch can process 258 Million packets per second (Mpps) with ∼100% accuracy for n=16384. In other words, SPArch can achieve a throughput of 132 Gbps with ∼100% accuracy for n=16384. Elastic Sketch has the lowest throughput among all the algorithms, while HeavyKeeper consumes the most memory. SPArch consumes less than half the memory of Elastic Sketch and ∼ 6× less memory than HeavyKeeper. The pipelining becomes laborious for Elastic Sketch as the latency in terms of the number of clock cycles varies with update scenarios, and pipelining may further increase the update latency. Fig. 27 depicts the throughput density in Mpps for ∼ 100% accuracy

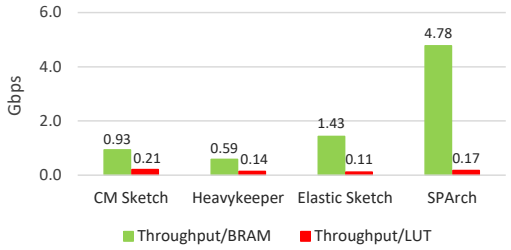

Fig. 27. Throughput versus memory and area

plotted against the area and memory utilization. SPArch achieves significantly higher throughput per BRAM, 3.34× than Elastic Sketch, thanks to its low memory requirement for achieving 100% accuracy. SPArch also has a higher throughput per area, 1.2× that of HeavyKeeper and 1.5× that of Elastic Sketch. However, CM Sketch exhibits a better throughput per area despite having a low throughput by virtue of its lowest area requirement.

9 PRACTICAL VALUE OF SPARCH

It is evident from the evaluation that SPArch outperforms other state-of-the-art architectures for the intended applications. However, it should be noted that SPArch is not a replacement for data structures such as Elastic Sketch or HeavyKeeper. SPArch is designed as a hardware-efficient network flow measurement/counter array architecture that can either perform specific standalone tasks such as flow measurement/heavy-hitter detection or can be integrated into other algorithms to efficiently perform versatile measurement tasks for other applications. While data structures like Elastic Sketch are designed for performing multiple detection tasks such as heavy-hitter detection, heavy change detection, and Cardinality Estimation, they are less hardware-efficient, not capable of leveraging the parallel computing capabilities of platforms like FPGAs, and can only perform sketch-based measurements.

Another practical value of SPArch is in unified DDoS detection tasks. With the increasing number of new and sophisticated attack patterns, it is required to integrate dedicated algorithms for accurate detection of a diverse range of attack vectors as a single algorithm cannot perform multiple attack detection tasks. Combining dedicated detection algorithms into a unified detection unit incurs substantial detection overhead and poses coordination challenges. Also, accurate and fast detection necessitates complex algorithms and more accurate flow measurements. For real-time implementation of such a unified detection unit, it is imperative that the algorithms share resource-intensive modules such as network flow measurement. The ability of SPArch to perform both sketch-based and eviction-based measurements, which other architectures cannot, along with its low memory requirement makes SPArch the best-suited architecture for implementing such shared measurement units.

While this paper primarily focuses on discussing and demonstrating the capabilities of SPArch as a counter array, its architectural design lends itself well to a wide range of other applications, paving the way for future research. One notable application is membership queries, a crucial aspect of network security applications. The drawbacks associated with associative arrays/CAM remain applicable to membership queries. By removing the counter array and retaining only the sketch part, SPArch can serve as an efficient alternative to a CAM, including the support for deletion. This architecture, during a query, can return the presence and the address of the element where it is located. Unlike CAM, SPArch does not store the elements themselves but offers

a virtual representation of storage. Furthermore, SPArch can seamlessly handle key-value store functionalities without introducing additional computational overhead or latency. SPArch also provides deletion support, similar to a CAM. A recent study [26] validates that a CAM with a depth of only 1024, designed to store a 96-bit element, consumes 6731 LUTs and 320 BRAMs, which is immense. In contrast, a SPArch key-value store that can store 1024 elements with near-hundred percent accuracy requires only 810 LUTs and 2 BRAMs. The above-mentioned results validate the advantage of SPArch as a replacement for CAM.

10 CONCLUSION

Probabilistic data structures are the most efficient components when it comes to memory consumption and speed, rendering them ideal for high-speed network flow measurements. The drawbacks of existing probabilistic flow measurement architectures made us rethink the way we design such probabilistic architectures in hardware implementations. In this work, we propose SPArch, a hardware-oriented sketch-based counter architecture for per-flow measurements. SPArch exploits the parallelism of FPGA and delivers high accuracy while consuming a minimal amount of memory. We evaluate SPArch in all aspects and compare it against the best available sketch-based measurement architectures, and the analysis validates the superior performance of SPArch compared to other architectures.

ACKNOWLEDGEMENTS

This work is supported by the ESCALATE project, funded by FWO under grant No. G0E0719N and SNSF under grant No. 200021L_182005, and by Cybersecurity Research Flanders under grant No. VR20192203.

REFERENCES

[1] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (Melbourne, Australia) *(IMC '10)*. Association for Computing Machinery, New York, NY, USA, 267–280.

[2] The Center for Applied Internet Data Analysis CAIDA. 2018. Passive OC48 and OC192 Traces. https://www.caida.org/data/passive/trace_stats/nyc-B/2018/?monitor=20181018-130000.UTC. Accessed: 2023.

[3] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Automata, Languages and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 693–703.

[4] Min Chen, Shigang Chen, and Zhiping Cai. 2017. Counter Tree: A Scalable Counter Architecture for Per-Flow Traffic Measurement. *IEEE/ACM Transactions on Networking* 25, 2 (2017), 1249–1262.

[5] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[6] CISCO. 2015. CISCO IOS NetFlow Version 9. http://www.cisco.com/c/en/us/products/ios-nx-os-software/netflow-version-9/index.html. Accessed: 2023.

[7] G. Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 55, 1 (2005), 58–75.

[8] Gil Einziger, Benny Fellman, and Yaron Kassner. 2015. Independent counter estimation buckets. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. 2560–2568.

[9] Cristian Estan and George Varghese. 2003. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.* 21, 3 (aug 2003), 270–313.

[10] The Canadian Institute for Cybersecurity (CIC). 2019. DDoS evaluation dataset (CIC-DDoS2019). https://www.unb.ca/cic/datasets/ddos-2019.html. Accessed: 2024.

[11] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. 2018. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 909–921.

[12] Mohamed Hassan. 2018. On the Off-Chip Memory Latency of Real-Time Systems: Is DDR DRAM Really the Best Option?. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. 495–505.

[13] Chengchen Hu, Bin Liu, Hongbo Zhao, Kai Chen, Yan Chen, Chunming Wu, and Yu Cheng. 2010. DISCO: Memory Efficient and Accurate Flow Statistics for Network Measurement. In *2010 IEEE 30th International Conference on Distributed Computing Systems*. 665–674.

[14] C. Hu, S. Wang, J. Tian, B. Liu, Y. Cheng, and Y. Chen. 2008. Accurate and Efficient Traffic Monitoring Using Adaptive Non-Linear Sampling Method. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*. 26–30.

[15] Nan Hua, Bill Lin, Jun (Jim) Xu, and Haiquan (Chuck) Zhao. 2008. BRICK: a novel exact active statistics counter architecture. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (San Jose, California) *(ANCS '08)*. Association for Computing Machinery, New York, NY, USA, 89–98.

[16] SunYoung Kim, Changhun Jung, RhongHo Jang, David Mohaisen, and DaeHun Nyang. 2021. Count-Less: A Counting Sketch for the Data Plane of High Speed Switches. *arXiv preprint arXiv:2111.02759* (2021).

[17] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 311–324.

[18] Yang Li, Hao Wu, Tian Pan, Huichen Dai, Jianyuan Lu, and Bin Liu. 2016. CASE: Cache-assisted stretchable estimator for high speed per-flow measurement. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications* (San Francisco, CA, USA). IEEE Press, 1–9.

[19] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 101–114.

[20] Y. Lu and B. Prabhakar. 2009. Robust Counting Via Counter Braids: An Error-Resilient Network Measurement Architecture. In *IEEE INFOCOM 2009*. 522–530.

[21] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China) *(VLDB '02)*. VLDB Endowment, 346–357.

[22] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory* (Edinburgh, UK) *(ICDT'05)*. Springer-Verlag, Berlin, Heidelberg, 398–412.

[23] Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (oct 1978), 840–842.

[24] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35, 3 (2015), 10–22.

[25] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented Sketch: Faster and More Accurate Stream Processing. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1449–1463.

[26] Arish Sateesan, Jo Vliegen, Joan Daemen, and Nele Mentens. 2020. Novel Bloom filter algorithms and architectures for ultra-high-speed network security applications. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 262–269.

[27] Arish Sateesan, Jo Vliegen, and Nele Mentens. 2022. An Analysis of the Hardware-Friendliness of AMQ Data Structures for Network Security. In *Security, Privacy, and Applied Cryptography Engineering: 12th International Conference, SPACE 2022, Jaipur, India, December 9–12, 2022, Proceedings* (Madras, India). Springer-Verlag, Berlin, Heidelberg, 287–313.

[28] Arish Sateesan, Jo Vliegen, Simon Scherrer, Hsu-Chun Hsiao, Adrian Perrig, and Nele Mentens. 2021. Speed Records in Network Flow Measurement on FPGA. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 219–224.

[29] Simon Scherrer, Jo Vliegen, Arish Sateesan, Hsu-Chun Hsiao, Nele Mentens, and Adrian Perrig. 2023. ALBUS: a Probabilistic Monitoring Algorithm to Counter Burst-Flood Attacks. In *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*. 162–172.

[30] Simon Scherrer, Che-Yu Wu, Yu-Hsi Chiang, Benjamin Rothenberger, Daniele E. Asoni, Arish Sateesan, Jo Vliegen, Nele Mentens, Hsu-Chun Hsiao, and Adrian Perrig. 2021. 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)*. 265–276.

[31] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A. Dinda, Ming-Yang Kao, and Gokhan Memik. 2007. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Trans. Netw.* 15, 5 (oct 2007), 1059–1072.

[32] sFlow. 2003. Traffic Monitoring using sFlow. http://www.sflow.org/sFlowOverview.pdf. Accessed: 2023.

[33] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 164–176.

[34] Lu Tang, Qun Huang, and Patrick P. C. Lee. 2020. A Fast and Compact Invertible Sketch for Network-Wide Heavy Flow Detection. *IEEE/ACM Trans. Netw.* 28, 5 (oct 2020), 2350–2363.

[35] Erez Tsidon, Iddo Hanniel, and Isaac Keslassy. 2012. Estimators also need shared values to grow together. In *2012 Proceedings IEEE INFOCOM*. 1889–1897.

[36] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. 1078–1086.

[37] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. 2016. Network-attached FPGAs for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*. 36–43.

[38] Hao Wu, Hsu-Chun Hsiao, and Yih-Chun Hu. 2014. Efficient Large Flow Detection over Arbitrary Windows: An Algorithm Exact Outside an Ambiguity Region. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 209–222.

[39] AMD Xilinx. 2023. Content Addressable Memory (CAM). https://www.xilinx.com/products/intellectual-property/efdi-cam.html. Accessed: 2023.

[40] Tong Yang, Siang Gao, Zhouyi Sun, Yufei Wang, Yulong Shen, and Xiaoming Li. 2019. Diamond Sketch: Accurate Per-flow Measurement for Big Streaming Data. *IEEE Transactions on Parallel and Distributed Systems* 30, 12 (2019), 2650–2662.

[41] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 561–575.

[42] Tong Yang, Lingtong Liu, Yibo Yan, Muhammad Shahzad, Yulong Shen, Xiaoming Li, Bin Cui, and Gaogang Xie. 2017. SF-sketch: A Fast, Accurate, and Memory Efficient Data Structure to Store Frequencies of Data Items. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 103–106.

[43] Yang Zhou, Hao Jin, Peng Liu, Haowei Zhang, Tong Yang, and Xiaoming Li. 2018. Accurate per-flow measurement with bloom sketch. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 1–2.

[44] Yang Zhou, Peng Liu, Hao Jin, Tong Yang, Shoujiang Dang, and Xiaoming Li. 2017. One Memory Access Sketch: A More Accurate and Faster Sketch for Per-Flow Measurement. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference* (Singapore). IEEE Press, 1–6.

[45] Haiting Zhu, Yuan Zhang, Lu Zhang, Gaofeng He, Linfeng Liu, and Ning Liu. 2020. SA Sketch: A self-adaption sketch framework for high-speed network. *Concurrency and Computation: Practice and Experience* 32, 23 (2020), e5891.