IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 13, 2021, accepted June 5, 2021, date of publication June 11, 2021, date of current version June 21, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3088500

A High-Throughput Hardware Accelerator for Network Entropy Estimation Using Sketches

JAVIER E. SOTO[1], PAULO UBISSE[1], YAIME FERNÁNDEZ[1], CECILIA HERNÁNDEZ[2,3], AND MIGUEL FIGUEROA[1], (Member, IEEE)

[1]Electrical Engineering Department, University of Concepción, Concepcion 4030000, Chile
[2]Computer Science Department, University of Concepción, Concepcion 4030000, Chile
[3]Center of Biotechnology and Bioengineering (CeBiB), Santiago 8370456, Chile

Corresponding author: Miguel Figueroa (miguel.figueroa@udec.cl)

This work was supported in part by the Agencia Nacional de Investigación y Desarrollo (ANID) Fondecyt Regular under Grant 1180995, in part by the Basal Funds under Grant FB0001, and in part by the Magíster Nacional and Doctorado Nacional Scholarships.

ABSTRACT Network traffic monitoring uses empirical entropy to detect anomalous events such as various types of attacks. However, the exact computation of the entropy in high-speed networks is a difficult process due to the limited memory resources available in the data plane hardware. In this paper, we present a method and hardware accelerator to approximate the empirical entropy of a large data set with high throughput and sublinear memory requirements. Our method uses streaming algorithms that exploit the fine-grained parallelism of existing hardware platforms for data plane processing, such as field-programmable gate arrays (FPGAs). The method uses sketches to compute the cardinality of the stream and the frequencies of the top-K elements on line, and then it estimates the contribution to the entropy of the rest of the stream assuming a simple uniform distribution for these elements. Implemented on a Xilinx UltraScale+ ZCU102 FPGA, the accelerator implements the method using only on-chip memory, with less than 50% resource usage. Tested on real network traces of up to 120 million packets and more than 5 million flows, the accelerator estimates the empirical entropy with less than 1.5% mean relative error and 21 μs latency, and supports a minimum throughput of 204 gigabits per second.

INDEX TERMS Empirical entropy, network monitoring, hardware acceleration, field-programmable gate arrays, streaming algorithms, sketches.

I. INTRODUCTION

Anomaly detection is a research problem in data mining that aims to analyze large datasets to identify elements, events, or observations that do not conform to an expected normal behavior pattern [1]. Applications of anomaly detection are key in numerous fields, including credit card fraud detection [2], network intrusion detection [3], and environment monitoring [4]. Many of these applications generate continuous, dynamic, and large volumes of data [5], which poses difficult computational challenges to the algorithms that perform the analysis. In particular, the efficient detection of anomalous behavior in data streams is important in high-speed networks, where it is necessary to process the data and react to changes in its distribution in real time [6].

The associate editor coordinating the review of this manuscript and approving it for publication was Rentao Gu.

Many techniques for anomaly detection have been proposed, most of them based on statistics, classification, clustering, and information theory. Statistics-based techniques assume a distribution or probability model of the data, and detect anomalies when the data behavior does not conform to the underlying model [7]. Classification-based techniques use labeled data instances to learn a classification model, and assign new data sequences to normal or anomalous classes [8]. Clustering is an unsupervised technique that groups similar data elements, such that anomalous data is either situated far away from the nearest cluster center, or belong to small or sparse clusters [9]. Techniques based on information theory assume that anomalous data produce irregular variations in the information content of the dataset. The information in the data is characterized using different metrics such as entropy, relative entropy, and Kolmogorov complexity [8]. Particularly, entropy-based approaches have become increasingly important in network

flow anomaly detection, due to their scalability, sensitivity, and low false-positive rates. Entropy-based techniques can also characterize different network traffic types using only a few packet-header fields [10].

Entropy is a measure of uncertainty of a random variable. The more random the values are in the observed distribution, the higher its entropy. Therefore, changes in the entropy of traffic features, e.g. source IP-address, destination IP-address, or destination port, signal unusual events in the network. Using entropy in network traffic analysis provides two significant benefits: Firstly, it increases detection sensitivity to anomalous events that may not manifest as abrupt changes in traffic volume. Secondly, using such traffic features provide additional diagnostic information into the nature of the anomalous events, e.g. distinguishing between worms, Distributed Denial of Service (DDoS) attacks, and scans, which is not available in volume-based anomaly detection [11], [12]. Although generalizations to the classic Shannon entropy, such as the Tsallis and Rényi formulations, have shown advantages in low-traffic scenarios [13]–[16], Shannon's entropy is still widely used in current traffic analysis and anomaly detection algorithms [1], [12], [17]–[23].

In high-speed network streams, computing the exact entropy value is difficult because of the limited memory and processing resources available on network devices. This task is computationally expensive because it requires maintaining a large number of counters, one for each distinct data element seen during the observation period, and accessing these counters with a throughput equivalent to that of the input stream. Moreover, some applications might require computing the entropy of network traffic that combine two or more packet features, thus increasing the number of distinct elements in the stream. In the worst case, the number of distinct elements is proportional to the number of packets observed during the period used to compute the entropy. In turn, the number of packets is proportional to the length of the observation period and the speed of the network link. The network throughput can reach 200 million packets per second on links that currently operate at 100 Gigabits per second (Gbps), and which plan to support 400 Gbps shortly [24].

To address this problem, researchers have proposed estimation techniques with sublinear memory requirements that enable hardware accelerators to process packet data in real time [18]. Dedicated hardware accelerators are important in applications that require high data throughput because they can process data with predictable performance at lower cost and power consumption than traditional programmable architectures [25]. However, the performance of a hardware accelerator is often damped by its bandwidth to off-chip memory, which limits the maximum computational throughput achieved by the datapath. In contrast, algorithms with low space requirements can be implemented using on-chip storage, which provides high aggregated data bandwidth and fine-grained parallelism. Thus, accelerators that use only on-chip resources can achieve higher data throughput compared to solutions that require frequent access to off-chip memory.

In this paper, we propose an algorithm and hardware accelerator to estimate the Shannon entropy of network traffic in standard network traces. Our approach uses a sketch-based streaming algorithm with sublinear memory requirements to estimate the probability of occurrence of each element in the data stream, and a hardware accelerator architecture that executes the algorithm at link speeds exceeding 200 Gbps. Our results show that we can compute a good estimate of the entropy of a network data stream using only the probability of the most frequent elements and adjusting a simple distribution to the rest of the stream. Moreover, we can compute a good estimate of the element frequencies and the cardinality of the dataset using sketches with sublinear memory. Consequently, the algorithm can be implemented on a Field-Programmable Gate Array (FPGA) using only on-chip resources. The main contributions of our work are:

- A method for estimating the entropy of flows in a network stream, which empirically computes the probability of the most frequent elements and adjusts a simple uniform probability distribution to the rest of the stream.
- A streaming algorithm that uses a sketch to estimate the frequencies of the data elements in the stream, a priority queue to store the most frequent elements and their frequencies, and a second sketch to estimate the cardinality of the dataset.
- A special-purpose hardware accelerator that implements the method described above using the sketches to estimate the probabilities of the data elements and compute the entropy of the stream. Implemented on a Xilinx Zynq ultra-scale ZCU104 FPGA, the accelerator achieves a throughput of 400 million network packets per second, which enables it to operate at network speeds of more than 204 Gbps.

The rest of the paper is organized as follows: Section II reviews related work and the theory behind the key elements of our method, Section III formalizes the problem and describes our proposed algorithms, Section IV describes the architecture of our hardware accelerator, and Section V evaluates our results using the accelerator and a reference software implementation. Finally, we present our conclusions and discuss future work in Section VI.

II. RELATED WORK

Intuitively, the entropy of a dataset measures the diversity or randomness of its data elements. The entropy attains its minimum value of zero when all the elements in the dataset have the same value, and its maximum value when all the elements are different [12]. In network traffic analysis, the dataset is composed of a sequence of packets acquired during a time interval, and the elements are network flows identified by a combination of packet fields such as source and destination IP addresses, layer 4 protocols such as Transmission

Control Protocol (TCP) or User Datagram Protocol (UDP), and source and destination ports [1]. The entropy is computed for one or more combinations of these features to detect traffic anomalies caused, for example, by a DDoS attack. Detection algorithms classify traffic as anomalous when its entropy deviates by more than a predetermined threshold from the statistics of normal traffic [26].

Galeano-Brajones *et al.* [19] proposed an entropy-based solution to detect and mitigate DoS and DDoS attacks in IoT scenarios using a stateful software-defined network (SDN) data plane. The detection algorithm computes the entropy of the source/destination IP and source/destination port fields of each packet. They acquire a set of traffic data and compute its entropy at regular intervals. For each set, they define a normal entropy range as $[\mu - \theta\sigma, \mu + \theta\sigma]$, where μ is the mean of the entropy of previous sets, σ is the standard deviation, and $\theta \in [0, 1]$ is a tuning parameter that sets the sensitivity of the algorithm. Komazec and Gajin [1] aggregate multiple traffic features and compute their entropy during each observation period. Their results show that entropy-based detection is useful for multiple types of attacks, but it is less effective for attacks that behave similarly to regular network traffic, such as SYN flood.

Because of the throughput requirements of high-speed links, hardware accelerators are often used to compute the entropy of network traffic at line rate. However, frequent access to off-chip memory severely limits the performance of the accelerator, therefore the design of algorithms to estimate entropy with low memory and computational resource usage is an active research area [12], [20], [23], [25], [27]–[29]. Lall *et al.* [12] presented two time- and space-efficient algorithms to approximate entropy, which are suitable for high-speed data links. The first algorithm, inspired by the work of Alon *et al.* [30] to estimate frequency moments, has strong theoretical guarantees on resource usage and estimation error. The second algorithm improves the precision of the estimation by processing high-frequency and low-frequency items separately. They sample the stream with a small probability and, if an item is sampled exactly once, they compute its entropy using the first algorithm. If an item is sampled more than once, they estimate its exact frequency in the stream. This sampling method is similar to the Sample and Hold algorithm [31], which keeps a precise counter for an item after it has sampled it at least once. Wand and Ding [20] used random subsampling to approximate the empirical entropy of items in a stream. They compute a bound for the estimation error and use it to construct confidence intervals of empirical entropy. They also propose algorithms to speed up ranking and filtering applications, which progressively sample the data until they can return the correct answer with specified probability.

Counting sketches provide an alternative to packet sampling to obtain approximate counts of items in a stream. A sketch is a compact probabilistic data structure that summarizes traffic statistics using fixed-size memory, while providing an error bound for its frequency estimations [32]. Bhuvanagiri and Ganguly [27] proposed an algorithm for estimating the entropy in data streams with low space requirements using sketch insertion and deletion operations. They use a hierarchical algorithm that samples over sketches to estimate the values used to compute the entropy. Clifford and Cosma [28] proposed a simple, unbiased sketch based on random linear projections drawn from a maximally-skewed stable distribution, which requires no tuning and has near-optimal space complexity. Lai *et al.* [33] use the CountSketch (CS) data structure [29] to estimate the entropy norm without using exact counting.

FlexSketchMon [23] is a framework that collects network flow information in the data plane to support various sketch-based algorithms for traffic monitoring and measurement tasks. The hardware data plane comprises a hash table that stores packet and byte counters for all flows in the observation period, a flow key table that stores identifiers for all observed flows, and a Bloom filter. They demonstrate their framework with sketch-based algorithms that use the information collected in the data plane to estimate network flow entropy and to detect super spreaders and heavy hitters. The heavy hitters are those elements in a stream whose frequency exceeds a predetermined threshold [34]. In our own work, Soto *et al.* [25] proposed a method to estimate the empirical entropy of the elements of a dataset using a counting sketch with sublinear memory. The method tracks the most frequent elements of the stream, and approximates the entropy of the dataset using only the estimated frequencies of these elements.

Several works in the literature partially estimate network traffic entropy in the data plane of network switches [17], [18]. The most well-established and widely-adopted data-plane programming language is P4 (Programming Protocol-independent Packet Processors), which focuses on reconfigurability in the field, protocol independence, and target-hardware independence [35]. While P4 substantially raises the level of abstraction to program applications on programmable switches, it does not support control loops, basic arithmetic operations such as division, logarithms and exponential functions, or any operations with floating-point numbers [22]. Lai *et at.* [21] present an entropy estimation algorithm implemented on programmable data-plane devices using P4. Their algorithm is based on a sketch designed for entropy estimation [28], where they replace the operations for random projections with fast lookups on precomputed tables in the match-action pipeline. Their implementation can estimate network traffic entropy at the 100 Gbps line rate of a Tofino switch. Recently, Ding *et al.* [22], proposed P4Entropy, a strategy to estimate network traffic entropy entirely on the data plane using P4. Their solution includes algorithms for logarithmic and exponential functions with real numbers to compute entropy on the switch without using an aggregator node. They show that P4Entropy achieves similar accuracy to existing solutions [36] without

constraining the number of packets in an observation interval or requiring ternary content-addressable memory (TCAM), which is a scarce resource in programmable switches.

FPGAs are often used to implement packet processing algorithms in the data plane. The FlexSkethMon framework discussed above [23] uses a NetFPGA-SUME board. They store the counter table in external quad-data-rate (QDR) static RAM and the flow key table on external DDR3 DRAM. The Bloom filter is implemented using on-chip memory and limits the number of flow key table updates. Considering a worst-case scenario of 64-byte minimum-size Ethernet frames, the board can process network traffic at a line rate of 96 Gbps. Lai *et al.* [33] proposed an estimation algorithm using a CountSketch data structure on a NetFPGA-10G board with a Xilinx Virtex 5 FPGA. Their architecture can hash a 32-bit input key and update the sketch in 3 clock cycles. With a clock frequency of 160 MHz, their solution can process traffic with a throughput of 30 Gbps. In Soto *et al.* [25], we presented a hardware accelerator that estimates the entropy of the most frequent elements in the network stream using a CM-CU sketch and a priority queue of 10240 elements. Implemented on a Xilinx Zynq UltraScale+ ZCU102 FPGA, the accelerator operates at 181 Gbps assuming minimum-size Ethernet packets.

FPGAs have also been used to compute other statistical properties of network traffic for anomaly detection [37]–[39]. Tong and Prasanna [37] modified the heavy-change detection algorithm proposed by Krishnamurthy *et al.* [40] to support online data stream processing. Heavy change corresponds to the elements in the stream that experience abrupt changes in frequency between two consecutive observation intervals [41]. They proposed a fully-pipelined sketch-based architecture on an FPGA to accelerate their algorithm. Implemented on a Xilinx Virtex-7 XC7VX690T FPGA, they achieve sustained data rates between 96 and 103 Gbps using different configurations. Later, the authors published another FPGA-based accelerator to detect heavy hitters and heavy change [38] using a CountMin and a K-ary sketch. Their implementation on a Xilinx Virtex Ultrascale XCVU440 FPGA operates at 150 Gbps for heavy-hitter detection and at 100 Gbps for heavy change. In previous work [39], we proposed an accelerator that uses sketches to detect heavy hitters on a Xilinx Kintex-7 XC7K325T FPGA. Using only on-chip memory, the accelerator supports network traffic at 100 Gbps for minimum-size Ethernet packets. The work was later extended to mine emergent k-mers on DNA sequence data using the frequencies of the heavy hitters [42].

III. METHODS

In this section, we first define the empirical and normalized entropy. Then, we outline our approach to estimate the entropy and describe the data structures and algorithms implemented in our hardware accelerator.

A. PROPOSED SCHEME

The mathematical definition of the Shannon empirical entropy of a data set is given by Eqn. (1):

$$H = -\sum_{i=1}^{N} \frac{m_i}{M} \log \frac{m_i}{M}, \tag{1}$$

where m_i is the number of occurrences of element i, M is the total number of elements in the set, and N is the maximum number of distinct elements in the set. Note that $M = \sum_{i=1}^{N} m_i$. The value of the entropy in Eqn. (1) is dependent on N, which makes it difficult to compare the entropy of datasets of different sizes. In those cases, it is more convenient to use the normalized entropy, which has a value between 0 and 1 and is defined as:

$$H_{norm} = \frac{H}{\log N}. \tag{2}$$

Computing the exact value of the entropy is challenging when processing a large number of elements. As seen in Eqn. (1), its definition requires computing the frequencies m_i of each element i, in addition to the values of M and N. We can compute the value of M by simply counting the elements as they arrive. However, computing the number of distinct elements N and the frequencies m_i for each of the elements in the sequence is not straightforward. In real-time network analysis applications, the exact computation of these values can be difficult, as it requires creating and maintaining a counter for every different element seen in the sequence, and updating these counters at the same speed at which packets are received on the data link. Our proposed solution provides a method for estimating the value of the empirical entropy using sublinear space. The solution includes the use of data structures and algorithms that enable the design and implementation of a hardware accelerator that operates with low memory usage and high throughput.

Our approach is based on the observation that, when computing the entropy of a large dataset, the most frequent elements provide a greater contribution to the entropy than elements with low frequencies. Our scheme consists of estimating the empirical entropy by separating it into two components: the first component accounts for the contribution of the top-K most frequent elements, and the second component summarizes the contribution of the rest of the elements, from $K+1$ to N in order of frequency. This is shown in Eqn (3):

$$H = -\left[\sum_{i=1}^{K} \frac{m_i}{M} \log \frac{m_i}{M} + \sum_{i=K+1}^{N} \frac{m_i}{M} \log \frac{m_i}{M}\right]. \tag{3}$$

Assuming a choice of K that captures elements with the highest frequencies, the second component will contain the elements with the lowest frequencies. Moreover, the variance of these frequencies is small compared to that of the top-K elements. We simplify the computation of m_i in the second component by assuming that the elements in this set

are uniformly distributed, thus m_i is a constant value for all $i \in [K-1, N]$.

Defining $L = \sum_{i=1}^{K} m_i$ as the total number of occurrences of the top-K elements, the total number of occurrences of the elements in the second component is $\sum_{i=K+1}^{N} m_i = M - L$. In addition, the number of distinct elements in the second component is $N-K$. Therefore, assuming a uniform distribution, we can estimate $m_i = \frac{M-L}{N-K}$ for $K < i \leq N$. Replacing these values in Eqn (3), we obtain the entropy estimation shown in Eqn. (4):

$$\hat{H} = -\left[\sum_{i=1}^{K} \frac{m_i}{M} \log \frac{m_i}{M} + \sum_{i=K+1}^{N} \frac{\left(\frac{M-L}{N-K}\right)}{M} \log \frac{\left(\frac{M-L}{N-K}\right)}{M} \right]. \tag{4}$$

Because $\frac{M-L}{N-K}$ is a constant, we simplify Eqn. (4) to obtain Eqn. (5), which is our empirical entropy estimation:

$$\hat{H} = -\left[\sum_{i=1}^{K} \frac{m_i}{M} \log \frac{m_i}{M} + \frac{M-L}{M} \log \left(\frac{M-L}{M(N-K)} \right) \right]. \tag{5}$$

Analogous to the exact normalized entropy, we define the normalized estimated entropy in Eqn. (6):

$$\hat{H}_{norm} = \frac{\hat{H}}{\log N}. \tag{6}$$

Eqn. (5) shows that our estimation method requires selecting the number of high-frequency elements K, and computing both the frequency count for each top-K element m_i and the total number of distinct elements in the input sequence N. The next section describes the data structures and algorithms used to estimate these values.

B. ALGORITHMS

In this section, we describe the data structures and algorithms for estimating the top-K most frequent elements with their corresponding frequencies, as well as the number of distinct elements in the input sequence. To estimate these values with high accuracy and performance, we use sketch-based data structures that require sublinear space and enable fine-grained parallel processing.

Algorithm 1 presents our general algorithm to estimate the normalized empirical entropy of a network stream. The algorithm processes the stream elements in one pass and uses *CardSketch* to estimate the number of distinct elements (cardinality) in the stream, *FreqSketch* to estimate the number of occurrences (frequency) of each element, and the priority queue PQ to store the frequencies of the top-K elements.

For each element e in the stream, the algorithm updates the state of the sketches. The algorithm also updates the contents of PQ using an estimate of the current frequency of e provided by *FreqSketch*. If e is already in PQ, its frequency is updated. Otherwise, e is inserted into PQ if the queue is not full. If PQ is full, e is inserted only if its frequency is higher than the lowest-frequency element, thus replacing that element in the queue. The maximum size of PQ is the user-provided parameter K_{pq}, therefore the queue maintains at most the K_{pq} elements with the highest frequencies.

Once the algorithm has finished processing the stream, it computes the terms $\sum_{i=1}^{K} \frac{m_i}{M} \log \frac{m_i}{M}$ and $L = \sum_{i=1}^{K} m_i$ using the contents of PQ. It also computes K by counting the elements in PQ. It is worth noticing that $K \leq K_{pq}$ only when the queue is not full and stores the frequencies of all the elements in the stream, that is, when $N \leq K_{pq}$. In that case, $K = N$ and Eqn. (5) reduces to Eqn. (3). However, in most practical cases $N \gg K_{pq}$, and $K = K_{pq}$.

Finally, the algorithm computes the second term of Eqn. (5) using the estimated value of N provided by *CardSketch*, and returns the normalized empirical entropy estimate $\hat{H}_{norm}$.

Algorithm 1 Normalized Empirical Entropy Estimation

1: **input:** *network stream, maximum priority queue size* K_{pq}, *frequency sketch dimensions* d *and* w, *cardinality sketch precision* p, *hash function* h
2: **output:** *normalized entropy* $\hat{H}_{norm}$
3: $L \leftarrow 0, K \leftarrow 0, M \leftarrow 0, \hat{H} \leftarrow 0$
4: *FreqSketch.initialize*(d, w, h)
5: *PQ.initialize*(K_{pq})
6: *CardSketch.initialize*(p, h)
7: {Processing network stream elements}
8: **foreach** element e **in** *network stream*
9: *CardSketch.update*(e)
10: *FreqSketch.update*(e)
11: *PQ.update*$(e$, *FreqSketch.estimate*$(e))$
12: $M \leftarrow M + 1,$
13: **end**
14: {Computing first term of $\hat{H}$ in Eqn. (5)}
15: **foreach** frequency m_i **in** PQ
16: $L \leftarrow L + m_i$
17: $K \leftarrow K + 1$
18: $\hat{H} \leftarrow \hat{H} + \frac{m_i}{M} \log \frac{m_i}{M}$
19: **end**
20: {Computing second term of $\hat{H}$ in Eqn. (5)}
21: $N \leftarrow$ *CardSketch.estimate*()
22: $\hat{H} \leftarrow \hat{H} + \frac{M-L}{M} \log \frac{M-L}{M(N-K)}$
23: $\hat{H}_{norm} \leftarrow \frac{\hat{H}}{\log N}$

1) SKETCH-BASED ALGORITHMS FOR FREQUENCY COUNTING

To count the number of occurrences of elements in the stream, we used existing sketch-based data structures and streaming algorithms frequently used to solve the heavy hitter problem. The heavy hitters are the elements in the input stream whose frequency is higher than a threshold [32], [43]–[46]. As seen in Algorithm 1, we also used a priority queue to store the top-K elements with their corresponding frequencies.

We evaluated three sketch-based algorithms: CountMin (CM) [47], CountMin with conservative updates (CM-CU) [46], and CountSketch (CS) [32]. The three algorithms use the data structure shown in Fig. 1. The structure consists

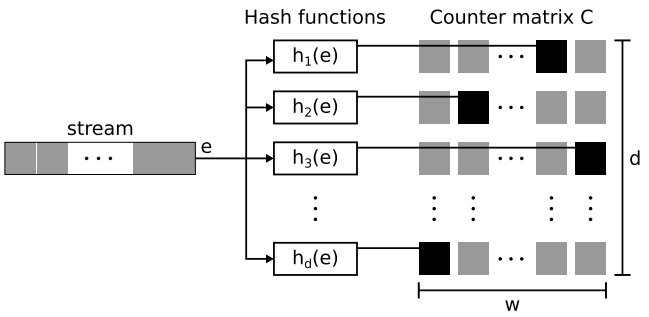


FIGURE 1. Sketch data structure to count elements in a stream. The sketch is a $d \times w$ matrix of counters C. The *update* and *estimate* operations map each element of the stream onto d counters, one in each row of C, using a set of hash functions.

of a matrix C with d rows and w columns, where each cell stores a frequency counter. The C matrix is initialized with zeros. For their *update* and *estimate* operations, the three algorithms use d hash functions to map an input element onto a column position in C for each of the d rows. The *update(e)* operation slightly differs in the three sketch algorithms: CM increments by one all of the d selected counters in C, CM-CU only increments the counters that contain the minimum value, and CS uses d additional hash functions to decide whether to increment or decrement each selected counter. The *estimate(e)* operation for CM and CM-CU returns the counter with the minimum value, whereas in CS it returns the median value of the d selected counters. Note that CS might under or overestimate element frequencies, but CM and CM-CU might only overestimate [47]. Goyal and Daum *et al.* [48] and Basat *et al.* [49] have compared CM to CM-CU and shown that CM-CU can, in practice, reduce overestimation compared to CM. Xiao *et al.* [50] evaluated CM-CU and found that it produces better frequency estimates than CM and CS. In our own previous work [39], [42], we show that, when identifying heavy hitters in a dataset, CM-CU produces more precise results with less memory usage compared to CM and CS. In Section V, we evaluate the three algorithms to estimate the frequency of the top-K elements, and find that our results are consistent with the literature mentioned above.

Algorithms 2-4 show the operations supported by the CM-CU sketch. Algorithm 2 shows the *initialize* operation, which sets all the counters in C to zero. Algorithm 3 shows the *update* operation, which maps the input element e onto d counters using the hash functions h_j, and computes the minimum value of the selected counters. It then increments those counters that hold the minimum value. Finally, Algorithm 4 shows the *estimate* operation, which also maps the input element e onto d counters and returns the minimum value.

2) SKETCH-BASED ALGORITHMS FOR CARDINALITY ESTIMATION

The space complexity of counting the exact number of distinct elements in a stream is linear in the cardinality of the set or the size of the stream. Therefore, in order to estimate cardinality with limited memory, several sketch-based

Algorithm 2 CM-CU Initialization

1: **input:** dimensions d and w, hash functions $h_j, j \in [1..d]$
2: $C[i][j] \leftarrow 0, i \in [1..d], j \in [1..w]$

Algorithm 3 CM-CU Update

1: **input:** stream element e
2: **for** $j = 1$ to d **do**
3: $\quad min\_est \leftarrow \min\{C[j, h_j(e)]\}, j \in [1, d]$
4: $\quad$ **if** $C[j, h_j(e)] = min\_est$ **then**
5: $\quad\quad C[j, h_j(e)] \leftarrow C[j, h_j(e)] + 1$
6: $\quad$ **end if**
7: **end for**

Algorithm 4 CM-CU Estimate

1: **input:** stream element e
2: **return** $\min\{C[j, h_j(e)]\}, j \in [1, d]$

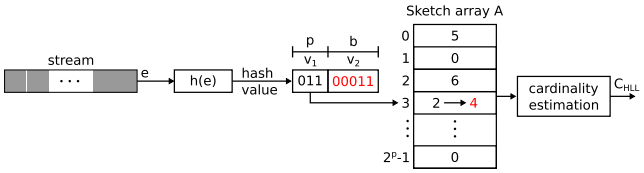


FIGURE 2. Hyperloglog sketch to estimate the cardinality of a stream. The sketch uses an array A of 2^p integers, where p is a precision parameter selected for the application. The *update* operation maps each input element of the stream onto a bucket of A using a hash function, and updates its value using Algorithm 6. The *estimate* operation computes the cardinality of the stream from the contents of A using Algorithm 7.

streaming algorithms have been proposed [51]–[53]. Among these, the HyperLogLog (HLL) sketch has sublinear space complexity and provides low estimation error in theory [53] and practice [54]. We use the HLL sketch in this work to estimate the number of distinct elements N in a network stream.

Figure 2 illustrates the operation of the HLL algorithm. It uses an array A of 2^p buckets, where each bucket contains an integer initialized to zero. The parameter p determines the size of the data structure, and is related to the precision of the cardinality estimation, as discussed later. For each new element e of the stream, the algorithm uses a hash function $h(e)$ to produce two values: a p-bit integer v_1, and a bit vector v_2 of length $b = |h(e)| - p$. The value of v_1 is used as an index to map the input element to a bucket in A. The algorithm then calculates the position of the leftmost one in v_2, counting from the most significant bit, and stores this value in the bucket $A[v_1]$ if it is higher than the value already stored in the bucket. In the example of Fig. 2, $v_1 = 011 = 3$ and the leftmost one in v_2 is in position 4. Before the update, $A[3] = 2$, and because $4 > 2$, the algorithm updates the value stored in the bucket to $A[3] = 4$. Note that the maximum value stored in any bucket in A is the number of leading zeroes in v_2 plus one. Therefore, the number of bits required for each bucket in A is $\lfloor \log(b) \rfloor + 1$. As discussed in Section V-B, our hardware

accelerator uses a 32-bit hash function with $p = 13$ and $b = 19$. Thus, each bucket in A is 5-bit wide.

After processing all the elements in the stream, HLL estimates its cardinality by first computing the harmonic mean Z of all the values in A as in shown in Eqn. (7):

$$Z = \sum_{i=0}^{|A|-1} 2^{-A[i]}. \tag{7}$$

Then, the algorithm estimates the cardinality C_{HLL} of the stream using Eqn. (8):

$$C_{HLL} = \alpha_A \frac{|A|^2}{Z}, \tag{8}$$

where α_A is a correction bias that depends of the number of buckets in the A array [53].

The space complexity of the HLL sketch is $\mathcal{O}(|A| \log \log N)$ and achieves a standard error of $1.03/\sqrt{|A|}$. Thus, increasing the number or buckets in A decreases the estimation error. However, for streams with low cardinality compared to $|A|$, most of the buckets in A are zero and the error increases. Heule *et al.* [54] show that, when $C_{HLL} \leq 2.5|A|$, the cardinality can be better estimated using Eqn. (9), where n_z is the number of zeros in A, as shown in Eqn. (9):

$$C_{HLL} = |R| \log \frac{|R|}{n_z}. \tag{9}$$

Algorithms 5-7 describe the initialization, update and estimation algorithms of the HLL algorithm. Algorithm 5 shows the *initialize* operation, which sets all the elements of A to zero and defines the value of the correction bias α_A when $|A| \geq 128$. For smaller values of $|A|$, α_A is a predefined constant [53]. Algorithm 6 shows the *update* operation, which computes the hash value x from the input element e, and uses the p most significant bits of x to select a bucket in A. It then uses the function *ldz()* to compute the position of the leftmost 1 in the b least-significant bits of x by counting the number of leading zeros, and updates the contents of the selected bucket in A. Algorithm 7 depicts the *estimate* operation, which computes the harmonic mean of the contents of A and uses it to estimate the cardinality of the stream.

Algorithm 5 HyperLogLog Initialization

input: precision parameter p, hash function h
Let $|A| = 2^p$
$\alpha_A = 0.7213/(1 + 1.079/|A|)$ **assuming** $|A| \geq 128$.
$A[i] \leftarrow 0, i \in [0, |A| - 1]$

Algorithm 6 HyperLogLog Update

input: network stream element e.
$x \leftarrow h(e)$
$v_1 \leftarrow < x_{31}, \cdots, x_{32-p} >_2$
$v_2 \leftarrow < x_{31-p}, \cdots, x_0 >_2$
$A[v_1] \leftarrow \max\{A[v_1], ldz(v_2) + 1\}$

Algorithm 7 HyperLogLog Estimate

$Z \leftarrow \sum_{i=0}^{|A|-1} 2^{-A[i]}$
$C_{HLL} \leftarrow \alpha_A \frac{|A|^2}{Z}$
if $C_{HLL} \leq 2.5|A|$ **then**
  $n_z \leftarrow CountZeros(A)$
  $C_{HLL} \leftarrow 2.5 \log(|A|/n_z)$
end if
return C_{HLL}

3) HASH FUNCTIONS

Sketch-based algorithms like CM-CU and HLL use hash functions to generate well-distributed hash values that reduce collisions when mapping input elements to their data structures. In our application, we use the 32-bit MurmurHash3 function described in Algorithm 8. Murmur3 is widely used in streaming algorithms because it is fast to compute and achieves low collision rates [55].

Algorithm 8 MurmurHash3

input: network stream element e, seed s
output: *hash value*
$k_0 \leftarrow$ `0xcc9e2d51` $\times$ e
$k_1 \leftarrow (k_0 \ll 15)|(k_0 \gg 17)$
$k_2 \leftarrow$ `0x1b873593` $\times k_1$
$k_3 \leftarrow k_2 \oplus s$
$k_4 \leftarrow (k_3 \ll 13)|(k_3 \gg 19)$
$k_5 \leftarrow 5 \times k_4 +$ `0xe6546b64`
$k_6 \leftarrow k_5 \oplus 4$
$k_7 \leftarrow k_6 \oplus (k_6 \gg 16)$
$k_8 \leftarrow$ `0x85ebca6b` $\times k_7$
$k_9 \leftarrow k_8 \oplus (k_8 \gg 13)$
$k_{10} \leftarrow$ `0xc2b2ae35` $\times k_9$
$k_{11} \leftarrow k_{10} \oplus (k_{10} \gg 16)$
return k_{11}

As shown in Algorithm 8, each instance of the function uses a randomly-chosen seed and it computes the hash value using a sequence of multiplications, additions, and bitwise logic operations such as or, xor and bit shifts.

IV. ACCELERATOR ARCHITECTURE

Figure 3 shows the architecture of our hardware accelerator to estimate the empirical entropy. The fields of interest extracted from the incoming network packets form the input stream, which is read by the accelerator. Each stream element e is processed by three principal blocks: the CM-CU sketch updates its frequency estimate for e, the HLL sketch updates its array, and a simple counter is incremented to compute the total number of elements M in the stream. The estimated frequency of e is also used to update the contents of the priority queue. Once the stream has been completely processed, the *Entropy* module computes the entropy estimate using Eqns. (5) and (6), where the frequencies of the top-K elements are extracted from the priority queue and the cardinality N of the stream

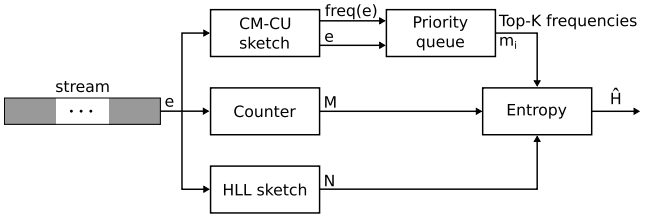


FIGURE 3. Hardware accelerator architecture. The accelerator receives a stream of packet attributes, which are processed by three blocks: a CM-CU sketch and priority queue to estimate the frequency of the top-K elements, a counter to compute the total number of elements of the stream, and an HLL sketch to estimate the number of distinct elements. The accelerator uses these values to estimate the empirical entropy of the network traffic using Eqns. (5) and (6).

is computed from the HLL sketch using Eqns. (8) and (9). The rest of this section describes the architecture of each of these modules.

A. COUNTMIN-CU SKETCH MODULE

Figure 4 shows the architecture of the CM-CU sketch when $d = 8$. The data structure is implemented as d memory blocks of w elements each and d parallel modules that compute a MurmurHash3 hash function using different seeds. The d memory blocks and hash-function modules are independent and operate in parallel. Each incoming element e, typically a network packet field such as the source IP address, is used as input to the d parallel MurmurHash3 modules. The resulting hash values are used to address a bucket in each of the d memory blocks to read the counter values associated with e in the sketch, as described in line 3 Algorithm 3. A pipelined comparator tree computes the minimum value of the d selected counters, which is then incremented by 1 and written back into the corresponding bucket, according to line 5 of Algorithm 3. The selected counter is also output as the frequency estimate of e in the stream, as specified by line 2 of Algorithm 4.

On an FPGA, we use on-chip parallel BRAM memory blocks to implement the sketch storage. These modules provide simultaneous read and write operations in a single clock cycle. The comparator tree and the MurmurHash3 modules are fully pipelined, thus the sketch can process one update operation per cycle. The BRAM blocks simultaneously read the counters associated to the current update and write back the new value of the counter selected by a previous update. Based on the analysis presented in Section V-C, the sketch parameters are $d = 8$ and $w = 16384$ in the current implementation of the accelerator.

The pipeline described above has a 5-cycle data hazard: when an update selects a counter to be incremented, the next 5 sketch accesses can potentially render an outdated estimate value if they happen to select the same counter. This is due to the latency of the pipeline, which is given by the one-cycle read operation, the 3-cycle latency of the comparator tree, and the one-cycle increment of the selected counter. We reduce the length of the hazard by incrementing all counters in parallel with the operation of the comparator tree, and by

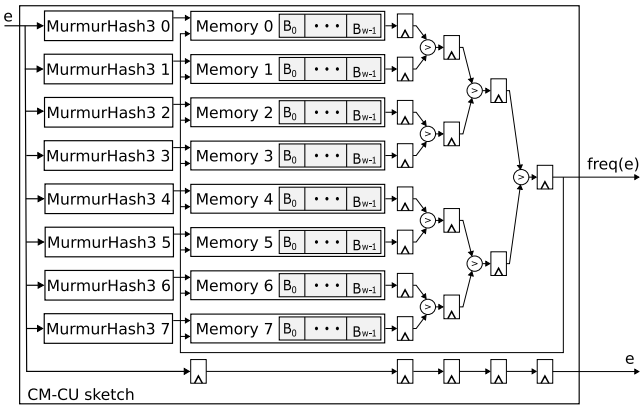


FIGURE 4. CM-CU sketch architecture with $d = 8$. The MurmurHash3 modules map the input to d buckets in the sketch. A comparator tree computes the minimum value of the buckets and outputs it as the frequency estimate. The counters stored in the buckets with the minimum value are incremented.

forwarding the updated counter value to the output of the memory block when we detect the hazard. However, a 3-cycle hazard remains in the pipeline. Based on our previous experience with similar designs [42], the probability of the hazard affecting the frequency estimate significantly is negligible, and we choose not to stall the pipeline to avoid it.

B. MurmurHash3 MODULE

The MurmurHash3 module implements the hash function in Algorithm 8. Our design is based on the 32-bit software version described in [55], and the seed is randomly chosen at design time for each instance of the module and treated as a constant during logic synthesis. Each line in Algorithm 8 is executed in a different clock cycle in a fully-pipelined fashion and the k_i values are stored in pipeline registers. Therefore, the MurmurHash3 module has a 12-cycle latency and a throughput of one new input per clock cycle. The module outputs a 32-bit result; the HLL sketch uses the entire hash value, but the CM-CU sketch uses only $\log w$ bits (14 bits in our current implementation) to address the BRAM blocks.

C. PRIORITY QUEUE MODULE

The method described in Algorithm 1 requires a priority queue (PQ) of K_{pq} elements to track the top-K elements of the stream. The accelerator must update the PQ every clock cycle to maintain the throughput achieved by the CM-CU sketch, which requires using K_{pq} independent registers to store the elements and keep them ordered. For large values of K_{pq}, this causes a severe impact in resource utilization and routing delay. To mitigate this problem, we propose an alternative architecture for the PQ that makes efficient usage of the BRAM memory blocks available in commercial FPGAs.

Figure 5 shows the architecture of our hardware PQ. The PQ is an array of S memory banks of size R with a shared address bus, such that $K_{pq} = R \times S$. The banks are implemented with BRAM resources on an FPGA. The PQ module uses this architecture to implement an array of R smaller PQs PQ_i of S elements each. When the CM-CU sketch outputs

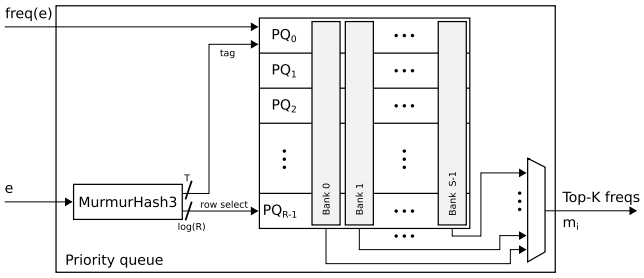


FIGURE 5. Architecture of the priority queue (PQ) module. The PQ is implemented as an array of R queues of S elements each. A MurmurHash3 function uniformly distributes the input elements across the R queues. The hash function also provides a tag to identify the input element in the PQ.

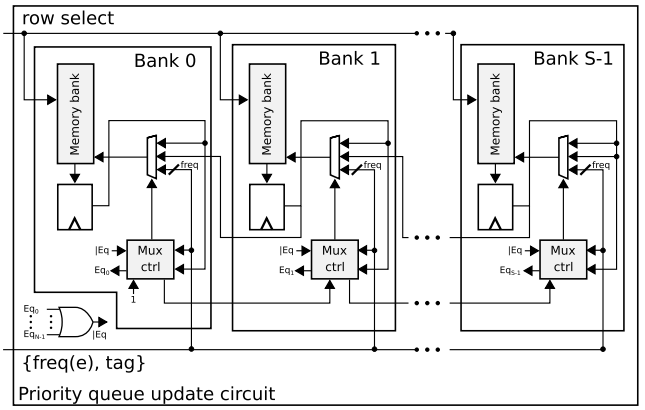


FIGURE 6. Update circuit for the priority queue. The queue uses the hash value of the input element to read S elements from the parallel memory banks. These values are stored in registers and operate as a sorted list. The tag field from the hash value is used to determine if the element is already in the queue and update its frequency. Otherwise, the input element is inserted into the queue if there is space available or if its frequency is higher than an element already in the queue.

a stream element e and its frequency $freq(e)$, the PQ computes a MurmurHash3 function using e as input. The $\log R$ least-significant bits of the hash value are used to access the priority queue PQ_i in the selected address. Then, e and its frequency are compared against the S entries in the selected PQ, and the contents of the queue are updated as described in Section III. To identify e in the queue, we use the T most significant bits of the hash value as a tag.

As described above, the architecture illustrated in Fig. 5 implements an array of R small PQs instead of a monolithic PQ of K_{pq} elements. However, because the MurmurHash3 distributes the stream elements uniformly across the R queues, this architecture captures the top-K elements in the stream with very good accuracy [25]. Moreover, Algorithm 1 only requires the frequencies of the top-K elements, so the order in which they are read is not relevant. Indeed, the contents of the PQ are only sorted within the smaller queues, which allows us to implement an update operation using a two-cycle latency pipeline with a throughput of one update per cycle. We use forwarding to avoid data hazards in the PQ update. As discussed in Section V-A, our current implementation uses $K_{pq} = 8192$, with $R = 1024$ and $S = 8$. Therefore, we use 10 bits to address the array and 22 bits for the tag. The elements in the PQ store a 22-bit tag and a 25-bit frequency count, and the memory banks are implemented with BRAM blocks on the FPGA.

Figure 6 shows the circuits that control PQ updates. The *row select* address from the hash value is used to read the S memory banks and store the values in local registers. These values are then compared in parallel to the incoming frequency and tag in their local *Mux ctrl* blocks. The S-element PQ behaves like a sorted list, with the lowest-frequency element in position 0. For each element i in the queue, $Mux\ ctrl_i$ indicates whether the input element is stored in this position in the queue (Eq_i) and whether the input frequency is lower than the stored value. A logic OR between the Eq_i signals determines whether the input element is already in the queue, in which case its frequency and position in the queue are updated. If the element is not in the queue, then it is inserted in the correct place unless its frequency is lower than the lowest-priority element. Inserting an element or updating its position in the queue requires shifting a subset of the elements in the queue by one position to the left. The *Mux ctrl* blocks determine whether to keep the current value, replace it with the input element, or to perform a left shift. The 3-input multiplexer in each bank selects the value to write in the memory bank: it selects between the current value, the input element, or the value stored in the cell adjacent to the right.

D. HYPERLOGLOG MODULE

As discussed in Section III-B2, we use an HLL sketch to estimate the cardinality of the stream. Figure 7 shows the architecture of the Hyperloglog module, which is divided into two main blocks: sketch updates and cardinality estimation. The sketch updates follow Algorithm 6: a MurmurHash3 block computes a 32-bit hash value from the input element e and the p most-significant bits are used to address the sketch array A and latch the stored value in a register. The *leading zeros* block computes the position of the leftmost one in the b least significant bits of the hash value. This position is compared to the value stored in the register, and the highest value is stored back into the array. According to Section V-B, $p = 13$ and $b = 19$. Thus, the array stores 8192 5-bit buckets, and is implemented using BRAM memory blocks on the FPGA.

The cardinality estimation block operates after the stream has been completely processed. It reads the elements of A, counting the number of zeros and computing line 1 of Algorithm 7. Because of the aspect ratio used to implement A in the accelerator, this process takes 1024 cycles. To compute Z, the block uses fixed-point arithmetic with 20 fractional bits: for each bucket in the array, the block shifts a 1 to the right based on the value in the bucket and accumulates the results. It then computes the reciprocal of Z using a Newton-Raphson algorithm with 10 iterations. Because the cardinality estimation is not in the critical path, this division is implemented using a nonpipelined finite-state machine in 40 clock cycles. Finally, the module multiplies the reciprocal of Z by $\alpha_A|A|^2$ to compute C_{HLL}.

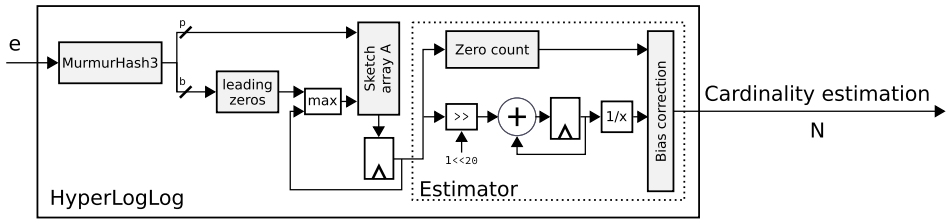


FIGURE 7. Architecture of the Hyperloglog module. The module includes an instance of the MurmurHash 3 block and an array that is updated for each element of the stream according to Algorithm 6. After the stream has been processed, the *Estimator* block computes the cardinality estimation using Algorithm 7. The circuit uses Newton-Raphson to compute the reciprocal of the harmonic mean of *A*, and a lookup table with linear interpolation to compute the logarithmic function in the bias correction.

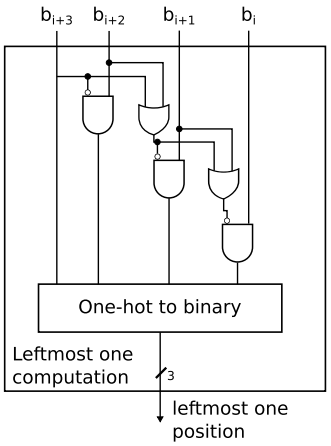


FIGURE 8. Most-significant one basic block. It uses a priority encoding circuit to output the position of the leftmost one in the input, or 0 if all the inputs are zero. By combining multiple circuits, we can compute the position of the leftmost one in an n-bit word, which is used in the HLL update, the Newton-Raphson divider, and the logarithmic function approximation.

The last block of the Hyperloglog module computes the bias correction in lines 3-5 of Algorithm 7 using the number of zeros n_z in A computed in the first block. We first compute $|A|/n_z$ using Newton-Raphson. To compute the logarithmic function, we first transform $|A|/n_z$ to a representation of the form $|A|/n_z = c \times 2^e$, where c uses 18 fractional bits and $1 \leq c < 2$. To compute the representation, the value of e is first determined by finding the position of the most significant 1 in $|A|/n_z$. Then, $\log(|A|/n_z)$ is computed as $\log c + e$ using a lookup table (LUT) with 1024 precomputed values. The 10 most significant bits of c address the LUT to obtain the first approximation, and the 8 remaining bits are used to linearly interpolate between the two adjacent values stored in the LUT.

There are three instances where the Hyperloglog module needs to compute the position of the most significant 1 in a number: the *leading zeros* block used in the HLL update, the Newton Raphson divider to obtain the first estimation of the reciprocal, and the computation of the logarithmic function. Figure 8 shows the basic block used to compute the leftmost one in an n-bit word. The block is a 4-input priority circuit that outputs a number between 1 and 4 to indicate the position of the most significant one, or zero if all the inputs are zero. To compute the position of the leftmost one of the 19-bit hash value in the sketch update, we use 5 priority circuits in parallel, each processing 4 bits of the input, and then compute the position based on the leftmost circuit whose output is not zero.

E. ENTROPY COMPUTATION MODULE

The final module in the accelerator uses the information stored in the priority queue, the total number of network packets M stored in the input counter, and the stream cardinality estimation N from the HLL sketch to estimate the network traffic entropy using lines 15-23 of Algorithm 1. Like the cardinality estimation block of the Hyperloglog module, the entropy computation module operates off line after the input stream has been completely processed by the other elements of the accelerator. Consequently, its performance does not impact the throughput of the accelerator and its operations are implemented sequentially.

First, the entropy module reads the contents of the PQ to obtain the frequency counts m_i for the top-K elements of the stream, which takes $K_{pq} = 8192$ clock cycles. To avoid computing the division m_i/M in line 18 of Algorithm 1, we take advantage of the fact that $log(a/b) = \log a - \log b$ to rewrite Eqn. (5) as Eqn. (10):

$$\hat{H} = -\frac{1}{M}\Big[(M-L)\big(\log(M-L) - \log(M(N-K)\big) - L\log M + \sum_{i=1}^{K} m_i \log m_i\Big]. \quad (10)$$

Following lines 16-17 of Algorithm 1, the entropy module reads the PQ, accumulates the values of m_i to compute the value of L, and uses a counter to compute K. The module also accumulates the values of $m_i \log m_i$. This process is performed in parallel with the operation of the cardinality estimation block that yields the value of N. We compute the logarithmic function and division using the same hardware described in Section IV-D. With the values of L, K, M and N, the module computes $\hat{H}$ and $\hat{H}_{norm}$ using Eqns. (10) and (6), respectively. The entire computation of the entropy estimate takes 59 clock cycles after reading the PQ contents, for a total latency of 8251 clock cycles.

V. RESULTS

This section describes the experiments performed to evaluate our method and hardware accelerator, and to set the parameters of the method. We also describe the network traces used in the experiments. First, we evaluate the estimation error of Eqn. (6) to determine the value of K. Next, we use software simulations of Algorithm 1 to choose the values of p and b in the HLL sketch for cardinality estimation, and of the counting sketch type and dimensions for frequency estimation. Then, we evaluate the estimation error incurred by our implementation of the accelerator, which includes the PQ array and the logarithm and division approximations. Finally, we evaluate the throughput, resource utilization, and power consumption of the hardware accelerator and discuss our results.

We use twelve different network traces in each of our experiments and evaluated our method by computing the entropy of the source IP-address field in each network packet. The traces used in our experiments are:

- One server trace from the Mendeley Data Research Network [56], a cloud-based repository for storing, sharing and finding data.
- Six traces from the Center for Applied Internet Data Analysis (CAIDA) [57]; five traces were collected at the Equinix-Chicago backbone link and one corresponds to the Equinix-San Jose link. The traces contain anonymized packet headers in the pcap format and have a duration of 60 seconds.
- Five 15-minute traces from MAWILab [58], a database that assists researchers to evaluate traffic anomaly detection methods.

Table 1 lists the traces, including the exact values for the total number of packets M, distinct source IP addresses N, and normalized entropy H_{norm} of each trace. As the table shows, the size of the traces spans a wide range: CAIDA provides the smallest datasets and MAWILab the largest, both in number of packets and distinct elements. The normalized entropy of the CAIDA traces is higher than the MAWILab sets.

In the rest of this section, we present our experiments, results, and selected parameters for our method and accelerator.

A. METHOD

To evaluate the results provided by our method, we use the absolute and relative estimation error defined in Eqns. (11) and (12):

$$E_{est} = |H_{norm} - \hat{H}_{norm}| \tag{11}$$

$$E_{r\_est} = \frac{|H_{norm} - \hat{H}_{norm}|}{H_{norm}} \times 100, \tag{12}$$

where H_{norm} is the exact normalized entropy of the trace given by Eqn. (2), and $\hat{H}_{norm}$ is our estimation of the normalized entropy given by Eqn. (6) using the exact values of the cardinality N and frequency counts m_i.

TABLE 1. Network traffic traces used in our experiments.

Dataset	M	N	H_{norm}
Mendeley	2,668,026	101,833	0.271
Chicago-20150219	15,962,528	252,239	0.593
Chicago-20160121	31,197,995	135,269	0.647
Chicago-20110608	27,046,414	393,237	0.694
Sanjose-20081016	20,919,376	334,579	0.701
Chicago-20080515	12,242,152	199,412	0.757
Chicago-20080319	3,938,619	167,768	0.804
Mawi-20201400	119,923,870	5,394,529	0.366
Mawi-20191400	62,478,145	4,633,485	0.406
Mawi-20181400	76,066,888	4,674,492	0.404
Mawi-20171400	115,486,661	4,145,741	0.437
Mawi-20161400	94,738,984	5,083,756	0.448

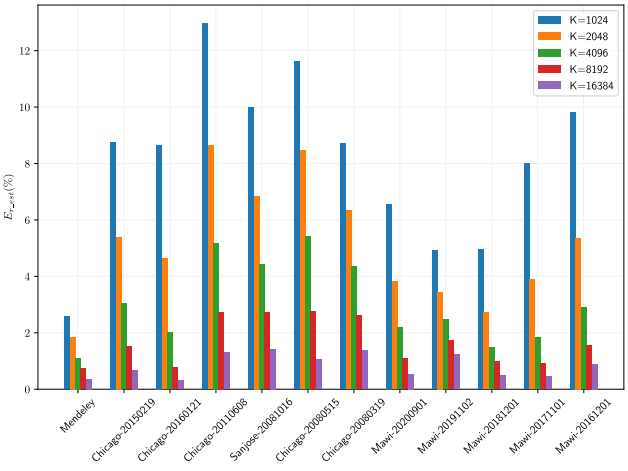


FIGURE 9. Relative entropy estimation error of Eqn. (6) with respect to Eqn. (2) as a function of K, considering the exact values of N and m_i.

In our first experiment, we evaluate the estimation error of our method as a function of K. Figure 9 shows the value of E_{r_est} for our proposed method using different values of K between 1024 and 16384. We use the exact frequency counts of the top-K elements and assume that frequency of the bottom $N-K$ elements is uniformly distributed. As expected, the estimation error decreases with the value of K. Moreover, for $K \geq 8192$, $E_{r\_est} < 3\%$ for all traces and its mean value is 1.67%. Considering that a normalized entropy error lower than 3% is sufficient to discriminate between the entropy of normal traffic and different types of DoS and DDoS attacks [22], we chose $K = 8192$. As discussed in Section IV-C, we implement the PQ as an array of $R = 1024$ PQs of $S = 8$ elements.

B. CARDINALITY SKETCH

Our second experiment evaluates the impact of the HLL sketch to estimate the cardinality of the trace. As discussed in Section III-B2, the performance of the sketch is determined by the precision parameter p, since $b = 32 - p$ for a 32-bit hash function. This parameter also determines the size of the sketch array, which is $A = 2^p$ buckets of $\lfloor \log(32-p) \rfloor + 1 \rfloor$ bits. In this experiment, we compute the estimated entropy using Eqn. (6) with $K = 8192$ and estimate N with an HLL

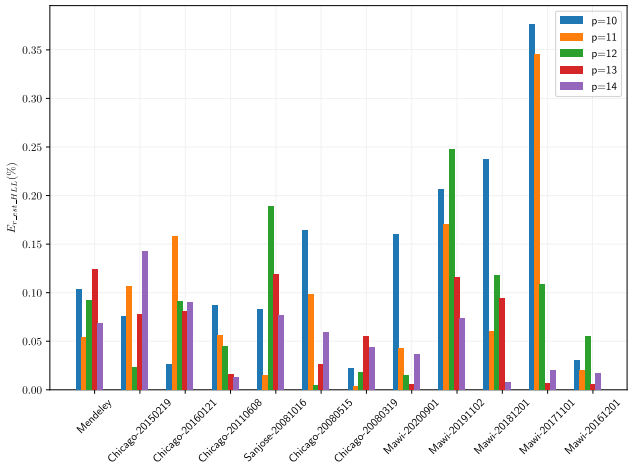


FIGURE 10. Relative error with respect to the entropy estimate of Eqn. (6) when using an HLL sketch to estimate N, as a function of the precision parameter p.

sketch using $10 \leq p \leq 14$. We use the relative error of the entropy estimation with respect to $\hat{H}_{norm}$ to evaluate the impact of the sketch, which is given by Eqn. (13):

$$E_{r\_est\_HLL} = \frac{|\hat{H}_{norm} - \hat{H}_{norm\_HLL}|}{\hat{H}_{norm}}, \tag{13}$$

where $\hat{H}_{norm\_HLL}$ is the normalized entropy computed by Eqn. (6) using the value of N estimated by the HLL sketch.

Figure 10 shows the value of $E_{r\_est\_HLL}$ as a function of p for our network traces using the 32-bit Murmur3 hash function described in Algorithm 8. In general, the entropy estimation using the sketch is better for larger values of p, and in all cases it is smaller than 0.4%. We use $p = 13$, with which the error introduced by the HLL sketch is less than 0.15% and the 8192×5-bit sketch can be implemented with a less than two BRAM memory blocks on an FPGA.

C. COUNTING SKETCH

We evaluate the use of a counting sketch to estimate the value of the frequency counts m_i in Eqn. (5). We consider three different sketches: CS, CM and CM-CU, and four sizes for each sketch, with $w \in \{8192, 16384\}$ and $d \in \{8, 16\}$. We evaluate the relative error that the counting and cardinality sketches add to our entropy estimate $\hat{H}_{norm}$ using $K = 8192$, as defined by Eqn. (14):

$$E_{r\_est\_s} = \frac{|\hat{H}_{norm} - \hat{H}_{norm\_s}|}{\hat{H}_{norm}}, \tag{14}$$

where $\hat{H}_{norm\_s}$ is the normalized entropy estimated by Eqn. (6) using the frequencies m_i estimated by the counting sketch and the cardinality N estimated by the HLL sketch with $p = 13$.

Table 2 shows the value of $E_{r\_est\_s}$ for all the traces in our experiment. The first column lists the trace and its estimated entropy $\hat{H}_{norm}$ using exact counts and $K = 8192$. The rest of the columns show E_{est_s} for each sketch type and size. The table shows that, for the parameter values chosen for this experiment, the precision of the sketches is highly dependent of the sketch size. In the case of CM and CM-CU, the dependency is stronger with respect to w, which determines the number of columns in the array, than d, which sets the number of rows. The count overestimation of CM results in entropy estimations that are in most cases less accurate than CS, and consistently worse than CM-CU, for the same sketch size. The estimation accuracy achieved with CM-CU is consistently better than CS and CM using the same size. Moreover, the CM-CU entropy estimation error obtained using $d = 8$ is close to the error with $d = 16$. Based on these results, we choose to use a CM-CU sketch with $w = 16384$ and $d = 8$, which adds an average of 0.2% to the estimation of Eqn. (6), and achieves the best compromise between accuracy and memory requirements.

TABLE 2. Relative entropy estimation error using Eqn. (6) as a reference, for three sketch types and four sizes.

Trace: H_{norm}	w	$E_{r\_est\_s}(\%)$					
		CS		CM		CM-CU	
		$d=8$	$d=16$	$d=8$	$d=16$	$d=8$	$d=16$
Mendeley	8K	1.97	0.81	6.46	4.54	1.28	0.88
$\hat{H}_{norm} = 0.273$	16K	0.84	0.38	1.66	1.10	0.45	0.33
Chicago-20150219	8K	8.97	1.81	7.13	4.78	0.55	0.26
$\hat{H}_{norm} = 0.602$	16K	2.04	0.41	1.90	1.39	0.15	0.12
Chicago-20160121	8K	1.39	1.44	2.75	1.49	0.31	0.13
$\hat{H}_{norm} = 0.652$	16K	1.68	0.28	0.58	0.36	0.10	0.09
Chicago-20110608	8K	10.10	3.14	9.72	5.81	0.64	0.15
$\hat{H}_{norm} = 0.713$	16K	3.85	0.59	2.28	1.58	0.08	0.04
Sanjose-20081016	8K	10.33	2.45	11.01	7.29	0.72	0.34
$\hat{H}_{norm} = 0.720$	16K	2.72	0.65	2.85	2.06	0.21	0.17
Chicago-20080515	8K	9.82	2.48	8.61	4.82	0.49	0.14
$\hat{H}_{norm} = 0.778$	16K	2.60	0.58	1.72	1.06	0.07	0.04
Chicago-20080319	8K	6.71	2.02	10.15	6.47	0.49	0.22
$\hat{H}_{norm} = 0.825$	16K	2.05	0.61	2.34	1.47	0.12	0.08
Mawi-20200901	8K	6.20	2.89	9.40	7.37	1.16	0.46
$\hat{H}_{norm} = 0.370$	16K	1.58	0.39	3.24	2.90	0.18	0.12
Mawi-20191102	8K	12.85	1.47	12.72	10.74	1.58	1.03
$\hat{H}_{norm} = 0.413$	16K	3.10	0.19	4.52	4.02	0.30	0.17
Mawi-20181201	8K	4.11	1.74	10.02	8.37	1.18	0.71
$\hat{H}_{norm} = 0.408$	16K	4.00	0.15	3.57	3.28	0.21	0.12
Mawi-20171101	8K	6.34	2.83	6.75	4.95	1.10	0.41
$\hat{H}_{norm} = 0.441$	16K	4.12	0.29	2.18	1.96	0.20	0.14
Mawi-20161201	8K	9.59	4.31	12.38	9.66	1.95	0.83
$\hat{H}_{norm} = 0.455$	16K	3.41	0.52	4.26	3.84	0.39	0.27

D. HARDWARE ACCELERATOR

Finally, we examine the impact of the implementation techniques used in the accelerator. These include replacing the conventional PQ of Algorithm 1 with the array of small PQs described in Section IV-C, which makes efficient use of the FPGA BRAM memory blocks, and the arithmetic function approximations described in Section IV-D. As discussed above, the accelerator uses an 8K-bucket HLL sketch with $p = 13$ and $b = 5$, a CM-CU sketch with $w = 16384$ and $d = 8$, and a PQ array with $R = 1024$ and $S = 8$. All three modules use 32-bit Murmur3 hash functions. The flow counters in the CM-CU sketch and the PQ are 25-bit wide, as required by the large MAWILab traces.

Table 3 summarizes our normalized entropy estimation results. $\hat{H}_{norm}$ is the entropy estimation computed with

TABLE 3. Summary of the entropy estimation obtained by Eqn. (6) and the accelerator.

Dataset	H_{norm}	$\hat{H}_{norm}$	$\hat{H}_{norm\_acc}$
Mendeley	0.271	0.273	0.272
Chicago-20150219	0.593	0.602	0.603
Chicago-20160121	0.647	0.652	0.650
Chicago-20110608	0.694	0.713	0.713
Sanjose-20081016	0.701	0.720	0.720
Chicago-20080515	0.757	0.778	0.780
Chicago-20080319	0.804	0.825	0.826
Mawi-20200901	0.366	0.370	0.364
Mawi-20191102	0.406	0.413	0.410
Mawi-20181201	0.404	0.408	0.405
Mawi-20171101	0.437	0.441	0.442
Mawi-20161201	0.448	0.455	0.453

TABLE 4. Summary of the absolute and relative estimation errors obtained by Eqn. (6) and the accelerator.

Dataset	H_{norm}	E_{est}	$E_{r\_est}(\%)$	E_{acc}	$E_{r\_acc}(\%)$
Mendeley	0.271	0.002	0.74	0.001	0.37
Chicago-20150219	0.593	0.008	1.35	0.010	1.69
Chicago-20160121	0.647	0.005	0.77	0.003	0.46
Chicago-20110608	0.694	0.019	2.74	0.019	2.74
Sanjose-20081016	0.701	0.019	2.71	0.019	2.71
Chicago-20080515	0.757	0.021	2.77	0.023	3.04
Chicago-20080319	0.804	0.021	2.61	0.022	2.74
Mawi-20200901	0.366	0.005	1.37	0.002	0.55
Mawi-20191102	0.406	0.006	1.48	0.004	0.99
Mawi-20181201	0.404	0.004	0.99	0.001	0.25
Mawi-20171101	0.437	0.004	0.92	0.003	0.69
Mawi-20161201	0.448	0.006	1.34	0.005	1.12

Eqn. (6), that is, the method estimation using exact counts and a conventional PQ to determine the frequencies of the top-8192 elements and the cardinality N of the stream. $\hat{H}_{norm\_acc}$ is the entropy estimation computed by the accelerator, and includes the hardware implementations of the PQ array and CM-CU sketch to compute the frequency of the top-8192 elements, and the HLL sketch to estimate N. Moreover, $\hat{H}_{norm\_acc}$ also includes the effect of the Newton-Raphson dividers and the computation of the logarithmic function using a LUT and linear interpolation.

Table 4 presents, for all 12 traces, the absolute and relative errors for the entropy estimation of our method, computed with Eqns. (11) and (12). It also presents the absolute and relative errors for the entropy estimated by the accelerator, computed as:

$$E_{acc} = |\hat{H}_{norm\_acc} - H_{norm}| \tag{15}$$

$$E_{r\_acc} = \frac{|\hat{H}_{norm\_acc} - H_{norm}|}{H_{norm}} \times 100, \tag{16}$$

where H_{norm} is the exact value of the normalized empirical entropy computed by Eqn. (2).

The relative estimation error of the method ranges between 0.74% and 2.77%, with a mean value of 1.65% and a standard deviation of 0.82%. The relative error in the Mendely trace is 0.74%, the mean error in the CAIDA traces is 2.16% and in the MAWILab traces it is 1.22%. In the accelerator, the estimation error ranges between 0.25% and 3.04%, with a mean of 1.45% and a standard deviation of 1.08%.

TABLE 5. FPGA resource utilization per module.

Module	LUTs	Registers	BRAMs	DSPs
Countmin-CU	2,854	2,991	128	96
PQ array	526	812	12	0
HLL	643	671	1.5	18
Entropy	643	844	0.5	14
Misc	31	87	0	0
Total	4,697	5,405	142	128
Available	230,400	460,800	312	1,728
Percentage (%)	2.04	1.17	45.51	7.41

In many cases, the error of the accelerator is actually lower than the method because the frequency errors introduced by the CM-CU sketch lower the entropy value, which the method consistently overestimates. The relative entropy error of the accelerator in the Mendely trace is 0.37%, the mean error in the CAIDA traces is 2.23%, and in the MAWILab traces it is 0.72%. Looking at the absolute entropy value, both the method and the accelerator show an estimation error lower than 0.023 for all traces. The mean absolute estimation error of the method is 0.010, and for the accelerator it is 0.009, with standard deviations of 0.008 and 0.009, respectively.

We modeled the accelerator architecture at the register-transfer level using the SystemVerilog hardware description language. We synthesized the design using Vivado 2018.2 and implemented it a Xilinx Zynq UltraScale+ ZCU102 FPGA. The design runs at 400 MHz, and the critical path is caused by a routing delay in the update operation of the CM-CU sketch. At this clock frequency, the accelerator can operate at a line rate of at least 204 Gbps, assuming the worst case of 64-byte minimum size Ethernet packets. The latency to compute the entropy after the last packet in the observation period is 21 μs.

Table 5 shows the hardware resource utilization of our implementation. The CM-CU sketch is the largest module: it uses 128 BRAMs (approximately 512 KB) to implement the sketch matrix. Even though the sketch uses 25-bit counters, the BRAM blocks can be configured only for 18 or 36 bits, thus each row of the sketch uses 16 $1K \times 36$-bit BRAMs. Most of the registers in CM-CU are used as pipeline registers in its 8 MurmurHash3 instances, and the DSP slices are used in the hash function operations, to increment sketch counters, and in the comparators that select the minimum value. The PQ array uses 12 BRAMs (approximately 48 KB) of storage, where each of the S memories uses 1.5 BRAMs to store 1024 tuples that contain the 25-bit counter and a 22-bit tag. Most of the individual registers are used to store temporary data in the pipeline. The queue shares the hash function with the first row of the CM-CU sketch, so it uses no DSP slices. The HLL module uses 1.5 BRAM for its array, and 18 DSPs for the Newton-Raphson dividers and the logarithmic function approximation. The entropy estimation module uses the remaining 14 DSPs on its dividers and logarithm computation. Overall, the entire design uses a fraction of the resources available on the chip, where BRAMs show the

TABLE 6. FPGA power consumption per module.

Module	Power (W)
Countmin-CU	0.90
PQ array	0.13
HLL	0.11
Entropy	0.09
Misc	0.01
Total	1.24

highest usage of 45.5%. Thus, significant hardware resources are available to expand the architecture or to add additional computation for network traffic analysis.

Table 6 shows the power consumption of the accelerator core reported by Xilinx Power Estimator. At 400 MHz, the core consumes 1.24 W, mostly at the BRAM blocks of the CM-CU sketch. The DSP slices also add to the power consumption, especially in the CM-CU and HLL modules.

E. DISCUSSION

Our results show that the estimation error is principally introduced by our characterization of the least frequent elements as a uniform distribution. Although the method overestimates the contribution of these elements to the entropy, it allows us to compute the entropy as a simple expression that depends on the information collected for the top-K flows in the network stream. Moreover, for a large enough K, the accelerator estimates the entropy with an error of 3% or lower, and an average of 1.45%, for the traces used in our experiments. The HLL and CM-CU sketches introduce a mean error to our estimation of 0.21%, the PQ array 0.06%, and the arithmetic function approximations add 0.46%. Using these techniques allow us to implement the complete entropy estimation algorithm on the data plane using exclusively on-chip resources, thus achieving high throughput, low power, and low resource utilization.

Other methods recently published in the literature estimate the entropy of network flows in the data plane. Lai *et al.* [21] used a sketch-based streaming algorithm programmed on a P4 switch. They tested their implementation on two sets of ten MAWILab traces from 2007 and 2015 and achieved mean estimation errors of 8.5% and 11.85%, respectively. Their sketch uses 2 MB of memory in the switch. Although using different traces makes it difficult to draw a direct accuracy comparison between these results and our work, when tested on traces of similar volume and cardinality (Chicago-20110608 and Mawi-20200901), our estimation error is lower than 0.6% using only 568 KB of on-chip memory. P4Entropy [22] and SOTA_entropy [36] develop algorithms for anomaly detection using P4 primitives in the data plane. Both use counting sketches to estimate the flow frequencies and incrementally compute the entropy. Although P4Entropy is less sensitive than SOTA_entropy to frequency overestimation by the sketch, they both achieve a similar error of 3% when using a CS sketch with CAIDA traces and observation intervals of 4 million packets [22]. As shown in Section V-D, we can estimate the entropy of network streams of more than 100 million packets with an error of 3% or lower.

Wellem *et al.* [23] use their FlexSketchMon framework, implemented on a NetFPGA-SUME board, to compute network flow entropy. In their solution, the data plane computes and stores counts for all flows in the observation period and the control plane computes the entropy. Because they keep exact counts, their precision is limited only by the on-chip Bloom filter used to control the updates to the flow key table. Using CAIDA traces with 15-second observation intervals and a Bloom filter of 512 KB or more, their entropy estimation error is consistently less than 0.4%. Comparatively, UnivMon [59], a sketch-based algorithm designed for P4 switches, estimates the entropy of the same traces with 3% error using 2 MB of memory. Although FlexMonSketch can estimate entropy with better precision than our method, the data structures used to maintain flow keys and flow counters are too large to be stored on-chip, therefore they use external QDRII+SRAM and DDR3 DRAM memory, respectively. Access to the counter table limits the throughput to 96 Gbps with 64-byte packets. Moreover, the size of the table is limited by the SRAM available on the NetFPGA board, and currently supports up to two million entries. In comparison, our accelerator uses only on-chip memory with less than one third of the BRAM blocks of FlexMonSketch, achieves a throughput of more than 200 Gbps and can process streams with more than 5 million flows.

VI. CONCLUSION

We have presented a method and the architecture of a hardware accelerator to estimate the entropy of network streams with sublinear memory and high throughput. The method only stores the frequencies of the top-K most frequent network flows and computes the entropy assuming a uniform distribution for the least frequent elements. Our streaming algorithm implements the method using sketches to estimate the flow frequencies and the cardinality of the data stream with sublinear memory, and a priority queue to store the top-K frequency counters.

The accelerator, implemented on Xilinx UltraScale+ ZCU104 FPGA, runs the estimation algorithm entirely on the data plane. It achieves a mean relative estimation error of less than 1.5% on network traces of more than 110 million packets and 5 million flows. The sublinear memory requirements of the algorithm enables it to run on the accelerator using only on-chip BRAM memory. As a result, the accelerator can process data at 204 Gbps assuming minimum-size Ethernet packets, it consumes 1.24 W of power and uses less than 50% of the memory and less than 10% of the arithmetic and logic resources available on the FPGA.

Our results show that the assumption of a uniform distribution for the least frequent elements accounts for most of our estimation error. Consequently, our current work focuses on improving the estimation of the distribution of these elements using the frequencies collected for the top-K flows. We are also working on integrating the

accelerator into network-specific hardware platforms such as the NetFPGA-SUME board, computing the Tsallis and Rényi entropies and other information-theory metrics, and designing classifiers that use these metrics to detect flow anomalies on the data plane.

REFERENCES

[1] T. Komazec and S. Gajin, "Analysis of flow-based anomaly detection using Shannon's entropy," in *Proc. 27th Telecommun. Forum (TELFOR)*, Nov. 2019, pp. 1–4.

[2] M. Raza and U. Qayyum, "Classical and deep learning classifiers for anomaly detection," in *Proc. 16th Int. Bhurban Conf. Appl. Sci. Technol. (IBCAST)*, Jan. 2019, pp. 614–618.

[3] W. Wu, Y. Huang, R. Kurachi, G. Zeng, G. Xie, R. Li, and K. Li, "Sliding window optimized information entropy analysis method for intrusion detection on in-vehicle networks," *IEEE Access*, vol. 6, pp. 45233–45245, 2018.

[4] I. Nevat, G. W. Peters, and I. B. Collings, "Distributed detection in sensor networks over fading channels with multiple antennas at the fusion centre," *IEEE Trans. Signal Process.*, vol. 62, no. 3, pp. 671–683, Feb. 2014.

[5] I. Souiden, Z. Brahmi, and H. Toumi, "A survey on outlier detection in the context of stream mining: Review of existing approaches and recommadations," in *Intelligent Systems Design and Applications*, A. M. Madureira, A. Abraham, D. Gamboa, and P. Novais, Eds. Cham, Switzerland: Springer, 2017, pp. 372–383.

[6] V. M. Tellis and D. J. D'Souza, "Detecting anomalies in data stream using efficient techniques: A review," in *Proc. Int. Conf. Control, Power, Commun. Comput. Technol. (ICCPCCT)*, Mar. 2018, pp. 296–298.

[7] J. Zhang, "Advancements of outlier detection: A survey," *ICST Trans. Scalable Inf. Syst.*, vol. 13, no. 1, pp. 1–26, 2013.

[8] A. Toshniwal, K. Mahesh, and R. Jayashree, "Overview of anomaly detection techniques in machine learning," in *Proc. 4th Int. Conf. I-SMAC (IoT Social, Mobile, Anal. Cloud) (I-SMAC)*, Oct. 2020, pp. 808–815.

[9] I. Souiden, Z. Brahmi, and H. Toumi, "A survey on outlier detection in the context of stream mining: Review of existing approaches and recommadations," in *Proc. Int. Conf. Intell. Syst. Design Appl.*, Porto, Portugal. Cham, Switzerland: Springer, 2016, pp. 372–383.

[10] G. Fernandes, J. J. P. C. Rodrigues, L. F. Carvalho, J. F. Al-Muhtadi, and M. L. Proença, "A comprehensive survey on network anomaly detection," *Telecommun. Syst.*, vol. 70, no. 3, pp. 447–489, Mar. 2019.

[11] J. Crichigno, E. Kfoury, E. Bou-Harb, N. Ghani, Y. Prieto, C. Vega, J. Pezoa, C. Huang, and D. Torres, "A flow-based entropy characterization of a NATed network and its application on intrusion detection," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2019, pp. 1–7.

[12] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *Proc. joint Int. Conf. Meas. Model. Comput. Syst. SIGMETRICS/Perform.*, vol. 34, no. 1, 2006, pp. 145–156.

[13] B. Tellenbach, M. Burkhart, D. Schatzmann, D. Gugelmann, and D. Sornette, "Accurate network anomaly classification with generalized entropy metrics," *Comput. Netw.*, vol. 55, no. 15, pp. 3485–3502, Oct. 2011.

[14] A. A. Amaral, L. D. S. Mendes, B. B. Zarpelão, and M. L. Proença, Jr., "Deep IP flow inspection to detect beyond network anomalies," *Comput. Commun.*, vol. 98, pp. 80–96, Jan. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366416306612

[15] C. Callegari, S. Giordano, and M. Pagano, "An information-theoretic method for the detection of anomalies in network traffic," *Comput. Secur.*, vol. 70, pp. 351–365, Sep. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404817301438

[16] K.-S. Yu, S.-H. Kim, D.-W. Lim, and Y.-S. Kim, "A multiple Rényi entropy based intrusion detection system for connected vehicles," *Entropy*, vol. 22, no. 2, p. 186, Feb. 2020. [Online]. Available: https://www.mdpi.com/1099-4300/22/2/186

[17] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "SketchVisor: Robust network measurement for software packet processing," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 113–126.

[18] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 561–575.

[19] J. Galeano-Brajones, J. Carmona-Murillo, J. F. Valenzuela-Valdés, and F. Luna-Valero, "Detection and mitigation of DoS and DDoS attacks in IoT-based stateful SDN: An experimental approach," *Sensors*, vol. 20, no. 3, p. 816, Feb. 2020.

[20] C. Wang and B. Ding, "Fast approximation of empirical entropy via subsampling," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 658–667.

[21] Y.-K. Lai, K.-Y. Shih, P.-Y. Huang, H.-P. Lee, Y.-J. Lin, T.-L. Liu, and J. H. Chen, "Sketch-based entropy estimation for network traffic analysis using programmable data plane ASICs," in *Proc. ACM/IEEE Symp. Architectures for Netw. Commun. Syst. (ANCS)*, Sep. 2019, pp. 1–2.

[22] D. Ding, M. Savi, and D. Siracusa, "Estimating logarithmic and exponential functions to track network traffic entropy in P4," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2020, pp. 1–9.

[23] T. Wellem, Y.-K. Lai, C.-Y. Huang, and W.-Y. Chung, "A flexible sketch-based network traffic monitoring infrastructure," *IEEE Access*, vol. 7, pp. 92476–92498, 2019.

[24] E. Viegas, A. Santin, A. Bessani, and N. Neves, "BigFlow: Real-time and reliable anomaly-based intrusion detection for high-speed networks," *Future Gener. Comput. Syst.*, vol. 93, pp. 473–485, Apr. 2019.

[25] J. E. Soto, P. Ubisse, C. Hernandez, and M. Figueroa, "A hardware accelerator for entropy estimation using the top-k most frequent elements," in *Proc. 23rd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2020, pp. 141–148.

[26] H. Hajji, "Statistical analysis of network traffic for adaptive faults detection," *IEEE Trans. Neural Netw.*, vol. 16, no. 5, pp. 1053–1063, Sep. 2005.

[27] L. Bhuvanagiri and S. Ganguly, "Estimating entropy over data streams," in *Proc. Eur. Symp. Algorithms*, Zürich, Switzerland. Berlin, Germany: Springer, 2006, pp. 148–159.

[28] P. Clifford and I. Cosma, "A simple sketching algorithm for entropy estimation over streaming data," in *Artificial Intelligence and Statistics*. Scottsdale, AZ, USA: PMLR, 2013, pp. 196–206.

[29] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theor. Comput. Sci.*, vol. 312, no. 1, pp. 3–15, Jan. 2004.

[30] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 137–147, Feb. 1999.

[31] C. Estan and G. Varghese, *New Directions in Traffic Measurement and Accounting*, vol. 32, no. 4. New York, NY, USA: ACM, 2002.

[32] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, Aug. 2008.

[33] Y. Lai, T. Wellem, and H. You, "Hardware-assisted estimation of entropy norm for high-speed network traffic," *Electron. Lett.*, vol. 50, no. 24, pp. 1845–1847, Nov. 2014.

[34] D. P. Woodruff, "New algorithms for heavy hitters in data streams (invited talk)," in *Proc. 19th Int. Conf. Database Theory (ICDT)*. Wadern, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[35] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[36] A. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time DDoS attack detection to programmable data planes," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, Apr. 2019, pp. 19–27.

[37] D. Tong and V. Prasanna, "High throughput sketch based online heavy change detection on FPGA," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2015, pp. 1–8.

[38] D. Tong and V. K. Prasanna, "Sketch acceleration on FPGA and its applications in network anomaly detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 929–942, Apr. 2018.

[39] A. Saavedra, C. Hernández, and M. Figueroa, "Heavy-hitter detection using a hardware sketch with the countmin-CU algorithm," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 38–45.

[40] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. ACM SIGCOMM Conf. Internet Meas. (IMC)*, 2003, pp. 234–247.

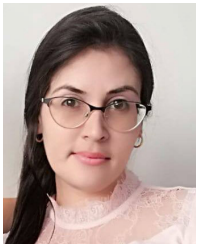
[41] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 741–756.

[42] A. Saavedra, H. Lehnert, C. Hernández, G. Carvajal, and M. Figueroa, "Mining discriminative K-mers in DNA sequences using sketches and hardware acceleration," *IEEE Access*, vol. 8, pp. 114715–114732, 2020.

[43] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. 28th Int. Conf. Very Large Databases (VLDB)*. Amsterdam, The Netherlands: Elsevier, 2002, pp. 346–357.
[44] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications," in *Proc. 4th ACM SIGCOMM Conf. Internet Meas. (IMC)*, 2004, pp. 101–114.
[45] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.
[46] A. Goyal, H. Daumé, III, and G. Cormode, "Sketch algorithms for estimating point queries in NLP," in *Proc. Joint Conf. Empirical Methods Natural Lang. Process. Comput. Natural Lang. Learn.* New York, NY, USA: ACM, 2012, pp. 1093–1103.
[47] G. Cormode and M. Muthukrishnan, "Approximating data with the count-min sketch," *IEEE Softw.*, vol. 29, no. 1, pp. 64–69, Jan. 2012.
[48] H. Daumé, III, and A. Goyal, "Approximate scalable bounded space sketch for large data NLP," in *Proc. Conf. Empirical Methods Natural Lang. Process.* Edinburgh, U.K.: Association for Computational Linguistics, 2011, pp. 250–261.
[49] R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik, "Faster and more accurate measurement through additive-error counters," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Jul. 2020, pp. 1251–1260.
[50] Q. Xiao, Z. Tang, and S. Chen, "Universal online sketch for tracking heavy hitters and estimating moments of data streams," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Jul. 2020, pp. 974–983.
[51] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, Oct. 1985.
[52] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Proc. Eur. Symp. Algorithms*, Budapest, Hungary. Berlin, Germany: Springer, 2003, pp. 605–617.
[53] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. 13th Conf. Anal. Algorithms (AoA)*, Juan-les-Pins, France, 2007, pp. 127–146.
[54] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proc. 16th Int. Conf. Extending Database Technol. (EDBT)*, Genoa, Italy, 2013, pp. 683–692.
[55] A. Appleby. (2012). *Smhasher & Murmurhash*. [Online]. Available: https://code.google.com/p/smhasher
[56] P. D. Bojovic, I. Basicevic, S. Ocovaj, and M. Popovic, "DDoS attack scoreboard dataset," , vol. 2017, data retrieved from Mendeley. [Online]. Available: https://data.mendeley.com/datasets/psjxnzsxyx/2.
[57] CAIDA. (2008). *The CAIDA UCSD Anonymized Internet Traces*. Data Retrieved From CAIDA. [Online]. Available: https://www.caida.org/data/passive/passive_dataset.xml
[58] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "MAWILab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking," in *Proc. 6th Int. Conf. (Co-NEXT)*, Philadelphia, PA, USA, 2010, pp. 1–12.
[59] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. ACM SIGCOMM Conf.* New York, NY, USA: ACM, Aug. 2016, pp. 101–114, doi: 10.1145/2934872.2934906.

JAVIER E. SOTO received the B.Sc. and M.Sc. degrees in electrical engineering from the Universidad de Concepción, Chile, in 2014 and 2016, respectively, where he is currently pursuing the Ph.D. degree in electrical engineering. His research interests include high-performance computing, hardware accelerators, embedded systems, and heterogeneous architectures.

PAULO UBISSE received the bachelor's degree in telecommunications and electronics engineering from Escola Superior de Ciências Náuticas, Maputo, Mozambique, in 2016. He is currently pursuing the M.Sc. degree in electrical engineering with the Universidad de Concepción. He is an Assistant Professor with Escola Superior de Ciências Náuticas. His research interests include digital systems, applications of microprocessors, and hardware accelerators for entropy estimation.



YAIME FERNÁNDEZ received the bachelor's degree in telecommunications and electronics engineering and the M.Sc. degree in telematics from Universidad Central Marta Abreu de Las Villas, Cuba, in 2013 and 2016, respectively. She is currently pursuing the Ph.D. degree in electrical engineering program with the Universidad de Concepción, Chile. Her current research interests include security in digital communications and hardware accelerators for scientific computing.

CECILIA HERNÁNDEZ received the M.Sc. degree in computer science from the University of Washington and the Ph.D. degree in computer science from Universidad de Chile. She is an Assistant Professor with Universidad de Concepción. Her research interests include compressed structures, data mining, parallel and distributed algorithms, and bioinformatics.

MIGUEL FIGUEROA (Member, IEEE) received the bachelor's degree in electronics engineering and the M.Sc. degree in electrical engineering from Universidad de Concepcion, Chile, in 1990 and 1997, respectively, and the M.Sc. and Ph.D. degrees in computer science and engineering from the University of Washington, Seattle, WA, USA, in 1999 and 2005, respectively. He is a Professor in electrical engineering with Universidad de Concepción. His current research interests include hardware accelerators for scientific computing and high-performance embedded systems, VLSI circuits for low-power video processing and computer vision, and high-performance instrumentation for quantum cryptography and radioastronomy.