# Estimating Logarithmic and Exponential Functions to Track Network Traffic Entropy in P4

Damu Ding[1,2], Marco Savi[1], and Domenico Siracusa[1]

[1]Fondazione Bruno Kessler, CREATE-NET Research Center, Trento, Italy
[2]University of Bologna, Bologna, Italy
{*ding, m.savi, dsiracusa*}*@fbk.eu*

***Abstract*—The evaluation of network traffic entropy is very useful for management purposes, since it helps to keep track of changes in network flow distribution. Nowadays, network traffic entropy is usually estimated in centralized monitoring collectors, which require a significant amount of information to be retrieved from switches. The advent of programmable data planes in Software-Defined Networks helps mitigate this issue, opening the door to the possibility of estimating entropy directly in the switches' data plane. Unfortunately, the most widely-adopted programming language used to program the data plane, called P4, lacks supporting many arithmetic operations such as logarithm and exponential function computation, which are necessary for entropy estimation. In this paper we propose two new algorithms, called P4Log and P4Exp, to fill this gap: these algorithms can estimate logarithms and exponential functions with a given precision by only using P4-supported arithmetic operations. Additionally, we leverage them to propose a novel strategy, called P4Entropy, to estimate traffic entropy entirely in the switch data plane. Results show that P4Entropy has comparable accuracy as an existing solution but without (i) constraining the number of packets in an observation interval and (ii) requiring the usage of TCAM, which is a scarce resource.**

## I. Introduction

For the purpose of management, network operators need to constantly monitor the status of the network and ensure that it behaves as intended. *Network traffic distribution* is an important indicator to understand the network behavior: the most widely-used metric to evaluate traffic distribution is *entropy* [1]. A periodical tracking of this metric helps diagnose performance and security issues, by supporting the execution of tasks such as congestion control [2], load balancing [3], port-scan detection [4][5], distributed denial-of-service (DDoS) attacks detection [6][7] and worm detection [8].

However, both in SNMP-based [9] legacy networks and in more recent Openflow-based [10] Software-Defined Networks, all these entropy-based monitoring applications are executed in a centralized component (generally known as *monitoring collector* or, more widely, *controller*) requiring the transmission, storage and processing of a huge amount of statistical information on active network flows from the network data plane [10]. This comes with two well-known drawbacks [11]: (i) a significant communication overhead is generated between data and centralized monitoring/control planes and (ii) significant processing capabilities are needed by the collector, with the risk of affecting performance of monitoring and network operations if involved parties are not well-dimensioned.

978-1-7281-4973-820$31.00 © 2020 IEEE

The recent advent of so-called (data-plane) *programmable switches* allows network operators to partially overcome those drawbacks. In fact, programmable switches can execute some network monitoring operations directly in their data plane pipeline (if appropriately programmed) and deliver to the centralized monitoring/control plane only processed information: potentially, this enables the computation of network traffic entropy directly in the switches' data plane and the sole forward of the resulted entropy value to the collector. However, data-plane programming comes with some inherent limitations: the most well-established and widely-adopted data-plane programming language, called P4 [12], does not allow the usage of loops (e.g. *for* or *while*). Additionally, it does not support some basic arithmetic operations such as division, logarithm and exponential function calculation, as well as any operation on floating numbers. Unfortunately, all these operations are needed for entropy computation. A straightforward solution to overcome these limitations could be executing all arithmetic operations unsupported by the P4 language at the collector, but this would almost nullify the benefits of the proposed solution as discussed so far.

The goal of this paper is to exploit P4 data-plane programming features to estimate network traffic entropy entirely in the switch data plane. As our network traffic entropy estimation strategy relies on the calculation of logarithm and division, we first propose a novel algorithm for the estimation of logarithm, called *P4Log*, that only uses arithmetic operations supported by the P4 language. Moreover, since division operation $\frac{A}{B}$ can be expressed as $2^{\log_2 A - \log_2 B}$, we propose another algorithm, called *P4Exp*, for the estimation of exponential functions with real-number exponent. When combined with P4Log, P4Exp can be used for the computation of divisions. Based on these two algorithms, we then present a novel strategy, named *P4Entropy*, to disclose network traffic distribution by leveraging *Shannon entropy* [13] computation. A prototype of P4Entropy has been implemented in P4 behavioral model [14] and has been proven to be fully executable in a P4 emulated environment.

We then evaluate P4Log, P4Exp and P4Entropy by means of simulations to show their effectiveness and their sensitivity to different tuning parameters. Results show that our algorithms can ensure similar relative error as state-of-the-art solutions

[6][15] that leverage on ternary Match+Action (M+A) tables to store some pre-computed values for estimation. The advantage of our strategies with respect to the state of the art is three-fold. First, our approach avoids the usage of any M+A table: this is especially beneficial to save memory consumption of TCAM (which is used to store ternary M+A tables but is limited, power-hungry and expensive). Additionally, our approach does not require any interaction with the control plane in executing the foreseen operations. Conversely, state-of-the-art solutions require that M+A tables are properly populated by a controller, generating some communication overhead: this is why we claim that our strategies work *entirely* in the data plane. Finally, our P4Entropy algorithm overcomes another limitation of the state-of-the-art benchmark strategy: while the existing strategy needs to set a fixed observation window on the number of processed packets to compute network traffic entropy, our approach allows to set as observation window any time interval, regardless of the number of processed packets. This is useful in the case estimated entropy values from multiple switches must be sent to the collector in a synchronized way.

The remainder of the paper is organized as follows. In Section II we report background notions. Section III describes P4Log and P4Exp, while Section IV describes P4Entropy. Sections V and VI present evaluation results and comparisons with existing solutions. In Section VII, we recall the related work. Finally, Section VIII concludes this paper and sets the future work.

## II. Background

In this section we recall background concepts needed to understand the strategies proposed in the following sections.

### A. Network traffic entropy

Network traffic entropy [1] gives an indication on traffic distribution across the network. Each network switch can evaluate the traffic entropy related to the network flows that cross it in a given time interval $T_{int}$. Relying on the definition of *Shannon entropy* [13], network traffic entropy can be defined as $H = -\sum_{i=1}^{n} \frac{f_i}{|S|_{tot}} \log_d \frac{f_i}{|S|_{tot}}$, where $f_i$ is the packet count of the incoming flow $i$, $|S|_{tot}$ is the total number of processed packets by the switch during $T_{int}$, $n$ is the overall number of distinct flows and $d$ is the base of logarithm. Traffic entropy reaches $H = 0$ when in $T_{int}$ all packets $|S|_{tot}$ belong to the same flow $i$, while it reaches its maximum value $H = \log_d n$ when each of the $n$ flows $i$ transports only one packet.

### B. Hamming weight computation

Hamming weight represents the number of non-zero values in a string. In a binary string, the Hamming weight indicates the overall number of ones. For example, given the binary string 01101, the Hamming weight is 3. Hamming weight can be computed by means of different algorithms: as part of P4Log, in this paper we adopt the *Counting 1-Bits* algorithm presented in [16], as it only relies on bitwise operations that are completely supported by P4 language [17].

TABLE I
OPERATIONS SUPPORTED BY P4 AND THEIR SYMBOLS [17]

| **Symbol** | **Operation** |
|---|---|
| $+$ | Addition |
| $-$ | Subtraction |
| $\cdot$ | Multiplication |
| $\gg$ | Logical right shift (Division by power of two) |
| $\ll$ | Logical left shift (Multiplication by power of two) |
| ˆ | Bitwise XOR |
| $\vert$ | Bitwise inclusive OR |
| & | Bitwise inclusive AND |
| $\sim$ | Bitwise COMPLEMENT |

### C. Binomial series expansion of exponential function $2^x$

The proposed P4Exp algorithm relies on binomial series expansion [18] of $2^x$, where $x$ is a real positive number. In general, the binomial series expansion of $(1+\alpha)^x$ is defined in the following way:

$$(1+\alpha)^x = \sum_{k=0}^{+\infty} \left( \begin{array}{c} x \\ k \end{array} \right) \alpha^k$$

When $\alpha$ = 1 we have the binomial series expansion of $2^x$:

$$2^x = \sum_{k=0}^{+\infty} \left( \begin{array}{c} x \\ k \end{array} \right) = \underbrace{1 + x + \frac{x(x-1)}{2!} + \cdots}_{N_{terms}}$$

With $\alpha = 1$, the series converges absolutely iff $\mathrm{Re}(x) > 0$ or $x = 0$. In our case this always holds, since $x$ is a real positive number. In P4Exp we will rely on a truncation of the binomial series to the first $N_{terms}$ terms.

### D. Sketch-based estimation of flow packet count

Estimating the number of packets for a specific flow crossing a programmable switch ($f_i$) is fundamental for network traffic entropy computation. Such an estimation can be performed by means of *sketches* [19], which are probabilistic data structures associated to a set of pairwise-independent hash functions. The *size* of each sketch data structure depends on the number of associated hash functions $N_h$ and on the output size of each function $N_s$, and is $N_h \times N_s$. *Update* and *Query* operations are used to store and retrieve information from the sketch: Update operation is responsible for updating the sketch to keep track of flow packet counts, while Query operation retrieves the estimated number of packets for a specific flow. Two well-known algorithms to Update and Query sketches are *Count-min Sketch* [20] and *Count Sketch* [21]. A detailed theoretical analysis on the accuracy/memory occupation trade-off for these sketching algorithms is reported in [20][21]. From a high-level perspective, as any of $N_h$ and $N_s$ increase, memory consumption is larger but estimation is more accurate. Count Sketch leads to a better accuracy/memory consumption trade-off than Count-min Sketch, but its update time is twice slower [22].

## III. Estimation of Log and Exp Functions in P4

In this section, we propose *P4Log* and *P4Exp*, two new algorithms for the estimation of logarithm and exponential function that leverage only arithmetic and logical operations supported by the P4 language (see Table I) and can be executed entirely in the data plane. The P4 code of two algorithms is open sourced in [23].

**Algorithm 1: P4Log algorithm**

```
Input: An $L$-bit integer $x$ ($L \in \{16, 32, 64, 128\}$) and a given
       logarithmic base $d$
Output: An $L$-bit integer estimation of $\log_d x$ amplified $2^{10}$
        times ($\log_d x \cdot 2^{10} \equiv \log_d x \ll 10$)
1  Function $log_2ES(x)$:
2      $w \leftarrow x|(x \gg 1)$
3      for int $i \in \{1, \ldots, \log_2 \mathrm{L} - 1\}$ do
4          $w \leftarrow w|(w \gg 2^i)$
5      $b \leftarrow HammingWeight(w)$
6      $n \leftarrow b - 1$
7      $\log_2 x \ll 10 \leftarrow n \ll 10 + \underbrace{\log_2(1 + \frac{\bar{x} - 2^{N_{bits}}}{2^{N_{bits}}}) \ll 10}_{\text{Tree search } \rightarrow N_{digits}}$
8      return $\log_2 x \ll 10$
9  Function $log_dES(x, \log_d 2 \ll 10)$:
10     $\log_d x \ll 10 \leftarrow (log_2ES(x) \cdot \log_d 2 \ll 10) \gg 10$
11     return $\log_d x \ll 10$
```

### A. P4Log algorithm

Given an $L$-bit integer $x$ and a logarithmic base $d$, the goal is to estimate $\log_d x$. Since operations on floating numbers are forbidden in P4, our algorithm computes $\log_d x$ amplified by $2^{10}$ times (i.e, $\log_d x \cdot 2^{10}$). This amplification (similar to what is done in [6]) is performed to deal with integer numbers without loosing accuracy on decimal parts, and can be done using the left shift operator ($\ll$). This is needed because P4, in the case of operations resulting in floating numbers, truncates the resulting value to its integer part: without any amplification our algorithm (as any other algorithm dealing with floating numbers) would result in a very bad estimation accuracy.

To compute $\log_d x$, we can write it as $\log_d x = \log_2 x \cdot \log_d 2$. Since $d$ is known, $\log_d 2$ is a constant value that can be pre-computed and loaded in the P4 program. Thus, logarithm estimation in P4 language reduces to the estimation of $\log_2 x$: if we can estimate $\log_2 x$, then it is always possible to estimate the logarithm of an input value $x$ with any given base $d$, as far as the constant value $\log_d 2$ has been stored in the P4 program as a constant.

As shown in Algorithm 1, the algorithm first estimates $\log_2 x$ ($log_2ES(x)$ function). Initially, it computes the integer part of $\log_2 x$, which is equal to the index of leftmost 1 of $x$ when expressed in binary notation. To get this information, all bits at the right of leftmost 1 of $x$ are iteratively converted to 1 and the result is stored in a binary string named $w$ (Lines 2 - 4). For instance, given a binary value 010010, the resulted $w$ is 011111. This operation is needed because, in P4, numbers are always handled in decimal notation, while we need a binary string as input of the next step. Then, the Hamming weight of $w$ (see Section II-B) is retrieved, indicating the number of bits from the leftmost 1 (including itself) and denoted by $b$ (Line 5). Hence, the index of the leftmost 1, called $n$ and equal to $b - 1$, stores the integer part of $\log_2 x$ (Line 6).

The algorithm then estimates the decimal part. Note that $\log_2 x = n + \log_2(1 + \frac{x - 2^n}{2^n})$, meaning that the estimation of the decimal part reduces to the estimation of $\log_2(1 + \frac{x - 2^n}{2^n})$. We adopt the first $N_{bits}$ bits starting from the leftmost 1 to estimate it, using a set of pre-computed decimal values stored in the P4 program as constants and rounded to a float with $N_{digits}$ digits of precision. If $N_{bits}$ bits are used to estimate the decimal part, it means that $2^{N_{bits}}$ constants need to be pre-computed and stored in the program. We call $\bar{x}$ the binary sub-string used to estimate the decimal part. For example, considering $N_{bits} = 2$, there are four possible cases: $\bar{x} \in \{100, 101, 110, 111\}$ (the leftmost 1 is always included in the sub-string). With $N_{digits} = 3$, each of the four cases leads to the following different estimations of the decimal part, computed as $\log_2(1 + \frac{\bar{x} - 2^{N_{bits}}}{2^{N_{bits}}})$:

$$\bar{x} = 100 \implies \log_2(1 + \frac{(000)_2}{(100)_2}) = \log_2(1 + \frac{0}{4}) = 0$$

$$\bar{x} = 101 \implies \log_2(1 + \frac{(001)_2}{(100)_2}) = \log_2(1 + \frac{1}{4}) \approx 0.322$$

$$\bar{x} = 110 \implies \log_2(1 + \frac{(010)_2}{(100)_2}) = \log_2(1 + \frac{2}{4}) \approx 0.585$$

$$\bar{x} = 111 \implies \log_2(1 + \frac{(011)_2}{(100)_2}) = \log_2(1 + \frac{3}{4}) \approx 0.807$$

The greater $N_{bits}$ and $N_{digits}$ are, the more accurate the estimation of the decimal part is. However, the bigger $N_{bits}$ is, the more pre-computed constants must be stored. Note also that, if $n < N_{bits}$, the algorithm performs zero padding.

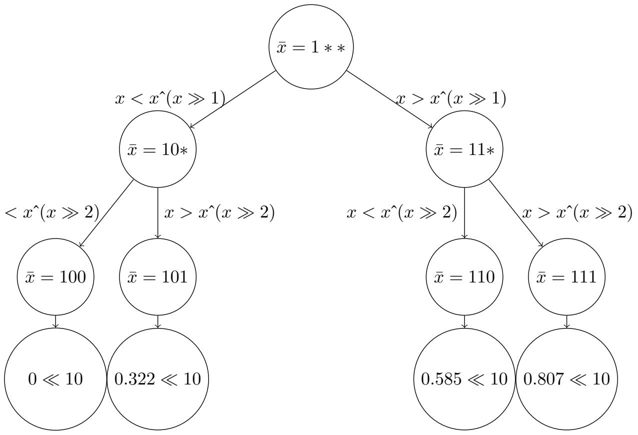


Fig. 1. Binary-tree data structure to extract the first $N_{bits} = 2$ bits of $\bar{x}$ and retrieve the estimated decimal part ($N_{digits} = 3$)

Unfortunately, for the same limitation of P4 recalled above, retrieving the $\bar{x}$ binary sub-string is not straightforward. However, by iteratively comparing $x$ with $x\hat{}(x \gg j)$, where $j \in \{1, \ldots, N_{bit}\}$ (integer), it is possible to obtain the string $\bar{x}$ and get the associated estimated decimal part. In fact, if $x < x\hat{}(x \gg j)$, it means that the $(j+1)$-th bit of $\bar{x}$ is 1, otherwise it is 0, being the first bit always 1 by definition. To this aim, we can define a binary tree in the P4 program by using $2^{N_{bits}} - 1$ if-else statements. An example of such a binary tree, in the case $N_{bits} = 2$ and $N_{digits} = 3$, is shown in Fig. 1. As shown in the figure, the constant decimal values are amplified $2^{10}$ times to ensure that they are integer numbers. Once the decimal part is retrieved, it is added to $n$ (also amplified $2^{10}$ times) to get an integer (amplified) estimation of $\log_2 x$. All these operations are summarized in Line 7 and 8 of Algorithm 1. Finally, the amplified estimated value of $\log_2 x$ (output of $log_2ES(x)$) is used to estimate

$\log_d x$ ($log_d ES(x, \log_d 2 \ll 10)$ function). The constant value $log_d 2$ is stored amplified $2^{10}$ times to prevent it being a floating number. For this reason, the result of $\log_2 x \cdot \log_d 2$, where both terms are amplified $2^{10}$ times, requires a division by $2^{10}$ to obtain an estimation of $\log_d x$ still amplified $2^{10}$ times. This can be done using the right-shift operator ($\gg$) (Lines 10-11).

### B. P4Exp algorithm

Given an $L$-bit integer $x$ and an exponent $d$, the goal is to estimate an integer approximation of $x^d$, with $d$ being any real number ($exp_d ES(x, d)$ function). Since $x^d = 2^{d \log_2 x}$, $x^d$ first requires the estimation of $\log_2 x$ by means of P4Log, and then the computation of $2^y$ where $y = d \log_2 x$. Our initial idea was to calculate $2^y$ by executing $1 \ll d \log_2 x$ in P4 language [17]. Unfortunately, in our case it is not possible to do so. In fact, the output of $log_2 ES(x)$ in Algorithm 1 is the estimation of $\log_2 x$ amplified $2^{10}$ times (to prevent accuracy losses) and $2^y$ cannot exceed $L$ bits ($2^L - 1$ is the biggest possible value), otherwise the computed number is set to 0 by P4. To ensure that the estimated $2^y$ value does not exceed $L$ bits, the exponent $y$ cannot be bigger than $\log_2 L$. Considering the biggest possible value for $L$, i.e., $L = 128$, the inequality $d \log_2 x \cdot 2^{10} < 128$ holds only in the case that $d \log_2 x < 2^{-3}$, which is still a too small value to make this approach meaningful.

To work around such a limitation, the algorithm decomposes $d \log_2 x = e_{int} + e_{dec}$, where $e_{int}$ is its integer part and $e_{dec}$ is its decimal part, meaning that $x^d = 2^{e_{int}} \cdot 2^{e_{dec}}$. P4Exp initially stores the result of $log_2 ES(x)$ (i.e., $\log_2 x \ll 10$) multiplied by $d$ in an integer variable called $exp \ll 10$ (Line 2). Note that the decimal part of the product is neglected and that this is the amplified version of the exponent. $e_{int}$ (not amplified) is then calculated by computing $exp \gg 10$, leveraging the limitation of P4 that a resulting floating number is always truncated to its integer part (Line 3). The algorithm then computes the amplified version of $e_{dec}$ as difference between the amplified versions of $exp$ and $e_{int}$ (Line 4). The estimated amplified version of $2^{e_{dec}}$ is retrieved by truncating its binomial series expansion to the first $N_{terms}$ terms (see Section II-C). All constants in the binomial series expansion need to be amplified by $2^{10}$ times. The inverse of the factorial number $v!$ in the binomial series can be estimated by $\lfloor \frac{2^{10}}{v!} \rfloor \gg 10$, where $\lfloor \frac{2^{10}}{v!} \rfloor$ is a pre-computed constant in the P4 program. For example, $\frac{1}{2!} = \frac{1}{2}$ can be computed by $\lfloor \frac{2^{10}}{2} \rfloor \gg 10 = 512 \gg 10 = \frac{1}{2}$. As $N_{terms}$ increases, more and more multipliers are amplified $2^{10}$ times, with the risk of going out of the $L$-bit range. Thus, the algorithm right-shifts 10 bits after each multiplication in the polynomial: this ensure that the resulted $2^{e_{dec}}$ is only amplified $2^{10}$ times. Lines 5-7 reports the estimation of $2^{e_{dec}} \ll 10$ with $N_{terms} = 3$.

Since computed values larger than $2^L - 1$ are set to 0, it must be ensured that $x^d$ will not be out of range for reasonable values of $d$. Thus, in the case $e_{int} < 10$ (small integer part), $x^d$ is estimated by calculating $2^{e_{int}}$ (not amplified and smaller than $2^{10}$), multiplying it by $2^{e_{dec}}$ (amplified as computed above) and dividing it by $2^{10}$, to get a non-amplified integer approximation (Lines 8-9). Conversely, for $e_{int} \geq 10$ integer parts, the algorithm compensates the $2^{10}$-times amplification of $2^{e_{dec}}$ by computing $2^{e_{int}-10}$ and multiplying it by the amplified version of $2^{e_{dec}}$ (Lines 10-11). Reducing the size of the exponent of $e_{int}$ by a factor of 10 helps prevent $x^d$ being out of range.

**Algorithm 2: P4Exp algorithm**

```
   Input: An $L$-bit integer $x$ (L $\in$ {16, 32, 64, 128}) and a given
          exponent $d$
   Output: An $L$-bit integer approximation of $x^d$
1  Function $exp_d ES(x, d)$:
2      $exp \ll 10 \leftarrow d \cdot log_2 ES(x)$
3      $e_{int} \leftarrow exp \gg 10$
4      $e_{dec} \ll 10 \leftarrow exp \ll 10 - e_{int} \ll 10$
5      $2^{e_{dec}} \ll 10 \leftarrow (1 \ll 10) + e_{dec} \ll 10$
6      $+(e_{dec} \ll 10 \cdot (e_{dec} \ll 10 - (1 \ll 10))$
7      $\gg 10 \cdot \lfloor \frac{2^{10}}{2!} \rfloor) \gg 10 + \underbrace{\cdots}_{\text{until } N_{terms}}$ (Binomial series expansion)
8      if $e_{int} < 10$ then
9          $x^d \leftarrow ((1 \ll e_{int}) \cdot (2^{e_{dec}} \ll 10)) \gg 10$
10     else
11         $x^d \leftarrow (1 \ll (e_{int} - 10)) \cdot (2^{e_{dec}} \ll 10)$
12     return $x^d$
```

## IV. Network traffic entropy estimation

Based on proposed P4Log and P4Exp algorithms in Section III, we propose a new strategy, named *P4Entropy*, to estimate the network traffic entropy entirely in the programmable switches' data plane. The prototype of P4Entropy has been implemented in P4 behavioral model [14] and is executable in an emulated environment as Mininet [24]. The source code is available in [25]. Formally, the problem is defined as follows.

**Problem definition:** Given a stream of incoming packets $S$ in a switch and a time interval $T_{int}$, returns *Shannon entropy* estimation (see Section II-A) at the end of $T_{int}$.

### A. Derivation of estimated entropy in P4

The goal of this section is to provide an estimation of network traffic entropy by only using P4-supported arithmetic operations and reducing as much as possible their number. The section also shows how relevant statistics, used for entropy estimation at the end of $T_{int}$, are iteratively updated every time a packet crosses the switch.

We first rewrite the Shannon entropy in the following way:

$$H(|S|_{tot}) = -\sum_{i=1}^{n} \frac{f_i(|S|_{tot})}{|S|_{tot}} \log_d \frac{f_i(|S|_{tot})}{|S|_{tot}}$$
$$= \log_d |S|_{tot} - \frac{1}{|S|_{tot}} \sum_{i=1}^{n} f_i(|S|_{tot}) \log_d f_i(|S|_{tot})$$

We consider $d = 2$ without any loss of generality. With respect to the definition given in Section II-A, we use the notation $f_i(|S|_{tot})$ to make explicit that $f_i$ refers to its value when $|S|_{tot}$ packets have been received (i.e., at the end of $T_{int}$). As packets arrive in the switch, the overall number of processed packets $|S|$ increases and must be stored in the switch to ensure that $H(|S|_{tot})$ can be computed at the end of $T_{int}$, when $|S| = |S|_{tot}$. We define $Sum(|S|) = \sum_{i=1}^{n} f_i(|S|) \log_d f_i(|S|)$, which must be updated as well. To understand how to update

$Sum(|S|)$, let's assume that a new packet for a specific flow arrives and is the $|S|$-th packet. We call its packet count $\bar{f}_i(|S|)$. It holds that:

$$\begin{cases} f_i(|S|) = f_i(|S|-1) & (f_i(|S|) \neq \bar{f}_i(|S|)) \\ f_i(|S|) = f_i(|S|-1)+1 & (f_i(|S|) = \bar{f}_i(|S|)) \end{cases}$$

This allows us to re-write $Sum(|S|)$ in the following way:

$$\begin{aligned} Sum(|S|) = {} & Sum(|S|-1) + \bar{f}_i(|S|)\log_2 \bar{f}_i(|S|) + \\ & -(\bar{f}_i(|S|)-1)\log_2(\bar{f}_i(|S|)-1) \end{aligned}$$

$Sum(|S|)$ thus needs two logarithmic computations for each incoming packet, and would require running P4Log twice with corresponding computational effort. In the next step, we show how it is possible to estimate $Sum(|S|)$ with only (at most) one logarithmic computation. When $\bar{f}_i(|S|) = 1$, we estimate $Sum(|S|) = Sum(|S|-1)$, being $\bar{f}_i(|S|)\log_2 \bar{f}_i(|S|) = 1\log_2 1 = 0$ and defining $(\bar{f}_i(|S|)-1)\log_2(\bar{f}_i(|S|)-1) = 0\log_2 0 = 0$ [26]. Instead, when $\bar{f}_i(|S|) > 1$, we need to re-write once again $Sum(|S|)$ in the following way:

$$\begin{aligned} Sum(|S|) = {} & Sum(|S|-1) + \log_2 \bar{f}_i(|S|) + \\ & + (\bar{f}_i(|S|)-1)\log_2(1 + \frac{1}{\bar{f}_i(|S|)-1}) \end{aligned}$$

According to L'Hopital's rule [27]:

$$\lim_{\bar{f}_i(|S|)\to+\infty} (\bar{f}_i(|S|-1)\log_2(1+\frac{1}{(\bar{f}_i(|S|-1)}) = \frac{1}{\ln 2}$$

Thus, we set $1/\ln 2 \approx 1.44$ as the approximation of the third term of $Sum(|S|)$. This approximation best works when most of the flows in $T_{int}$ carry a number of packets much greater than 1 (as usually happens in an ISP backbone network, which is the most suitable scenario where to apply our strategy). Finally, $Sum(|S|)$ can be estimated as:

$$Sum(|S|) \approx \begin{cases} Sum(|S|-1) & (\bar{f}_i(|S|) = 1) \\ Sum(|S|-1) + \log_2 \bar{f}_i(|S|) + 1/\ln 2 & \\ & (\bar{f}_i(|S|) > 1) \end{cases} \quad (1)$$

This estimation requires at most one logarithm computation.

Since P4 language does not support division, we re-write $\frac{1}{|S|_{tot}} = 2^{-\log_2 |S|_{tot}}$. So, entropy can be written as:

$$H(|S|_{tot}) = \log_2 |S|_{tot} - 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})}$$

In this form, entropy can be estimated by only using P4-supported operations, leveraging P4Log and P4Exp algorithms. In the following, we show how it is possible to further slightly reduce complexity in entropy estimation.

When $|S|_{tot} = \sum_{i=1}^{n} f_i(|S|_{tot}) > Sum(f_i|S_{tot}|)$, it holds that $0 < 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})} < 1$. This is a corner case that happens only when flow distribution is almost uniform (i.e., when most of flows carry only one or very few packets). In this case, we neglect the computation of $2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})}$, meaning that we estimate entropy as flow distribution was perfectly uniform. Network traffic entropy can then finally be estimated as follows:

$$H(|S|_{tot}) \approx \begin{cases} \log_2(|S|_{tot}) & (|S|_{tot} > Sum(|S|_{tot})) \\ \log_2(|S|_{tot}) - 2^{(\log_2 Sum(|S|_{tot}) - \log_2 |S|_{tot})} & \\ & (|S|_{tot} \leq Sum(|S|_{tot})) \end{cases} \quad (2)$$

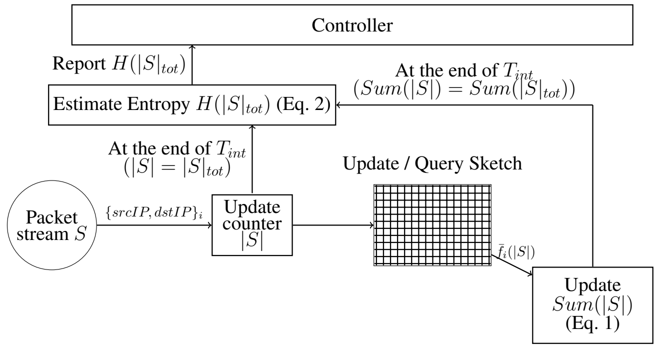


Fig. 2. Scheme of P4Entropy

**Algorithm 3: P4Entropy algorithm**

```
Input: Packet stream $S$, time interval $T_{int}$
Output: Entropy estimation $H(|S|_{tot})$ of $S$ in $T_{int}$
$|S| \leftarrow 0$
$Sum(|S|) \leftarrow 0$
Function UpdateSum:
    while currentTime $< T_{int}$ do
        for Each received packet belonging to flow $i$ do
            $|S| \leftarrow |S| + 1$
            $\bar{f}_i(|S|) \leftarrow Sketch(\{srcIP, dstIP\}_i)$
            if $\bar{f}_i(|S|) > 1$ then
                $Sum(|S|) \ll 10 \leftarrow Sum(|S|) \ll 10$
                $+log_2ES(\bar{f}_i(|S|)) + 1.44 \ll 10$
    $Sum(|S|_{tot}) \leftarrow (Sum(|S_{tot}|) \ll 10) \gg 10$
    return $Sum(|S|_{tot})$, $|S|_{tot}$
Function EstimateEntropy($Sum(|S|_{tot})$, $|S|_{tot}$):
    if currentTime = $T_{int}$ then
        if $|S|_{tot} > Sum(|S|_{tot})$ then
            $H(|S|_{tot}) \ll 10 \leftarrow log_2ES(|S|_{tot})$
        else
            diff $\leftarrow log_2ES(Sum(|S|_{tot})) - log_2ES(|S|_{tot})$
            $H(|S|_{tot}) \ll 10 \leftarrow log_2ES(|S|_{tot}) - exp_dES(2, \text{diff})$
    return $H(|S|_{tot}) \ll 10$
```

### B. P4Entropy algorithm

Figure 2 and Algorithm 3 show the scheme and pseudocode of P4Entropy algorithm, leveraging outcomes from Section IV-A. First, the algorithm continuously updates $Sum(|S|)$ until the end of $T_{int}$ (*UpdateSum* function) with flow information from incoming packets. A counter $|S|$ is used to count all incoming packets in the switch. We consider as *flow key* the source IP-destination IP pair of the packet, with $i \sim \{srcIP, dstIP\}_i$. However, other flow definitions could be considered (e.g. 5-tuple) without any loss of generality. A sketch data structure (e.g., Count Sketch or Count-min Sketch, see Section II-D) is used to store the estimated packet count for all the flows, being continuously updated to include information from new packets, and then it is queried to retrieve the estimated packet count $\bar{f}_i(|S|)$ for the flow $i$ the current incoming packet belongs to. This value is then passed to a readable and writable stateful register named $Sum(|S|)$, which is updated as specified in Eq. 1. All the floating numbers in the equation must be amplified $2^{10}$ times, since P4Log outputs an

TABLE II
DEFAULT PARAMETERS FOR P4LOG AND P4EXP

| Alg | Parameter | Value |
|---|---|---|
| P4Log | Digits of precision for decimal part ($N_{digits}$) | 3 |
| | Number of bits for estimation of decimal part ($N_{bits}$) | 4 |
| P4Exp | Digits of precision in $log_2ES(x)$ ($N_{digits}$) | 3 |
| | Number of bits in $log_2ES(x)$ ($N_{bits}$) | 7 |
| | Number of terms in binomial series ($N_{terms}$) | 7 |

amplified integer value. Only at the end of $T_{int}$, $Sum(|S|_{tot})$ is reduced by a factor of $2^{10}$ and its final value, together with $|S|_{tot}$, is returned (Lines 1-12 of the pseudocode).

Traffic entropy is then estimated as specified in Eq. 2. The resulted value of $H(|S|_{tot})$ is amplified $2^{10}$ times since output values of P4Log are amplified, while output values of P4Exp are not. Finally, the switch reports the amplified entropy estimation value to the controller, which can reset all the switch registers to start another estimation in the next $T_{int}$.

## V. EVALUATION OF P4LOG AND P4EXP

We implemented P4Log and P4Exp in Python for evaluation and sensitivity analysis, reported in this section.

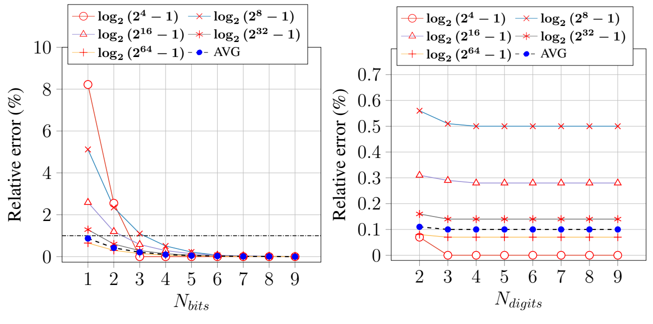
### A. *Evaluation metrics and settings*

*1) Metrics:* We consider *relative error* as key metric. For P4Log, given an input value $x$ and base $d$, relative error is defined as $\frac{|log_d ES(x, \log_d 2) - \log_d x|}{\log_d x} \cdot 100\%$, where $\log_d x$ is the exact value. For P4Exp, given input base $x$ and exponent $d$, the relative error is defined as $\frac{|exp_d ES(x,d) - x^d|}{x^d} \cdot 100\%$. In both cases, we consider as acceptable target a relative error of 1%, as also done in previous work [15].

*2) Default experimental settings:* Unless otherwise specified, the default tuning parameters in all experiments are the ones reported in Table II.

*3) Testing values for P4Log:* When using $N_{bits}$ to estimate the logarithm in our P4Log algorithm, all the bits after them are ignored and considered as 0. Intuitively, the algorithm leads to worst-case estimations when most significant bits after $N_{bits}$ are 1s. To make it as general as possible, we choose five different $l$-bit-length (i.e., $l = 4, 8, 16, 32, 64$ bit) input values where all bits are 1s (and thus the respective decimal value is $2^l - 1$). Moreover, we always consider $d = 2$ as logarithmic base since, as shown in Section III-A, different bases only require the multiplication of $\log_2 x$ with the constant value $\log_2 d$, and this operation does not affect the relative error to the exact value. We also randomly select $5 \cdot 10^6$ integer numbers such that $x \in \{1, 2^{64} - 1\}$ and average the relative error in logarithm estimation, named *AVG* in the following. With $5 \cdot 10^6$ randomly-selected number, 95% confidence-interval width of relative error is always smaller than 0.01%, and we do not plot it in shown graphs since it would overlap with the plotted markers.

*4) Testing values for P4Exp:* In this case, we evaluate the performance of the algorithm when both base and exponent of $x^d$ vary. We choose input values according to the following rules: (i) to evaluate the impact of base variation, we fix a 64-bit integer base $x$ to a chosen value, then we find the integer exponent $d$ that maximizes the output $x^d$ within 64 bits (we call this test *chosen-base variation*); (ii) to evaluate the impact of exponent variation, we fix the integer exponent $d$ to a chosen value, then we find the largest 64-bit integer base $x$ that maximizes the output $x^d$ within 64 bits (we call this test *chosen-exponent variation*); (iii) we select $5 \cdot 10^6$ integer numbers with base $x \in \{1, 2^{32} - 1\}$ and exponent $d \in \{2, 32\}$ both randomly chosen (these ranges ensure that $x^d$ is always within 64 bit) and average the relative error in exponential function estimation, named *AVG* in the following. Also in this case, 95% confidence interval has a width smaller than 0.01% and is not plotted in the graphs.



(a) (b)

Fig. 3. Sensitivity of P4Log to $N_{bits}$ (a) and $N_{digits}$ (b)

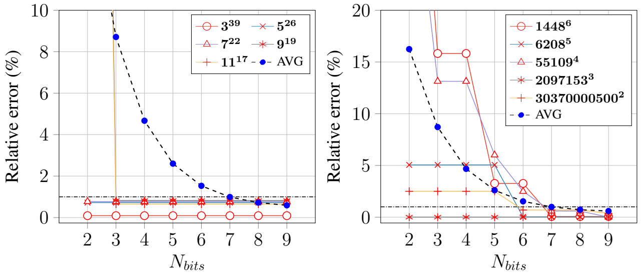
### B. *Evaluation of P4Log*

Figure 3 shows the sensitivity of P4Log with respect to a variation of $N_{bits}$ and $N_{digits}$. As shown in Fig. 3(a), the relative error of $\log_2 x$ decreases as $N_{bits}$ increases, with more significant improvement when the input value $x$ is small. When $N_{bits} = 4$, the relative error is below 1% in all the considered cases, becoming almost 0 when $N_{bits} = 6$. Instead, Fig. 3(b) shows that (i) an increase of $N_{digits}$ does not improve much the relative error, (ii) all relative errors are below 1% and (iii) when $N_{digits} = 4$ the relative error reaches its minimum. The *AVG* curve always shows an average relative error below 1%.

### C. *Evaluation of P4Exp*

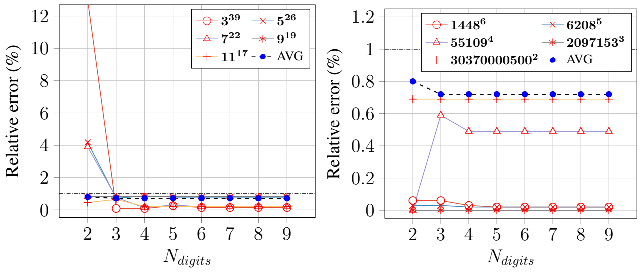
We analyze the sensitivity of P4Exp with respect to a variation of $N_{bits}$ and $N_{digits}$ of $Log_2ES(x)$ used for exponential estimation (see Algorithm 2) and to a variation of $N_{terms}$.

*1) Sensitivity to $N_{bits}$:* As shown in Fig. 4(a), in the case of chosen-base variation, when $N_{bits} \geq 3$ all the relative errors of considered $x^d$ are under 1%. When $N_{bits}$ is too small ($N_{bits} = 2$) a very large relative error (around 80%) is experienced for bases $x \geq 9$. This is because a small $N_{bits}$ causes a bad estimation of $d \cdot log_2ES(x)$ that is exponentially amplified when computing $2^{d \cdot \log_2 ES(x)}$. Such a bad estimation is especially evident when $d \cdot log_2ES(x)$ is large. Figure 4(b) shows instead that exponential functions with small exponent $d$ but large base $x$ are more sensitive to $N_{bits}$ and, as shown also above, the relative error decreases as $N_{bits}$ increases. When $N_{bits} = 7$ relative errors of all estimations are below 1%. The *AVG* curve shows that, on average, a large relative error is experienced when $N_{bits}$ is small and that $N_{bits} \geq 7$ ensures an average error below 1%.



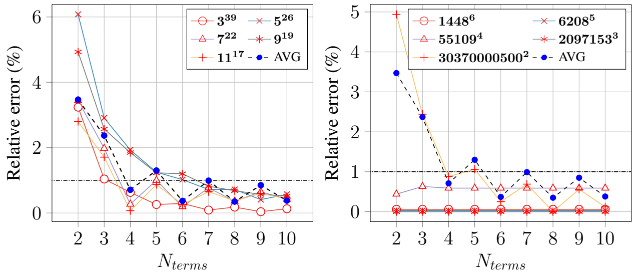
(a) Chosen-base variation (b) Chosen-exponent variation

Fig. 4. Sensitivity of P4Exp to $N_{bits}$



(a) Chosen-base variation (b) Chosen-exponent variation

Fig. 5. Sensitivity of P4Exp to $N_{digits}$



(a) Chosen-base variation (b) Chosen-exponent variation

Fig. 6. Sensitivity of P4Exp to $N_{terms}$

*2) Sensitivity to $N_{digits}$:* As shown in Fig. 5(a), in the case of chosen-base variation (and thus bigger exponents), for $N_{digits} \geq 3$ the relative error reaches values below $1\%$. Instead, Fig. 5(b) shows that computations involving smaller exponents but bigger bases are not very sensitive to $N_{digits}$, being relative errors always under $1\%$, and that considering bigger $N_{digits}$ may in some cases even prove counterproductive. The *AVG* curve shows a similar trend: on average, an increase in $N_{digits}$ does not affect much performance, and the average relative error is always below $1\%$.

*3) Sensitivity to $N_{terms}$:* Figure 6(a) shows that $N_{terms}$ strongly affects the estimation of $x^d$ especially when the exponent is large and the base is small. Relative errors oscillate but, in a long term, decrease as $N_{terms}$ increases. Oscillation is due to the way how binomial series converges. When $N_{terms} \geq 7$ relative error for all estimations is under $1\%$. Instead, as shown in Fig. 6(b), with small chosen exponents and large bases, relative errors are all below $1\%$ when $N_{terms} \geq 6$. Oscillation is also well visible in the *AVG* curve that shows how $N_{terms} \geq 6$ leads to an average relative error below $1\%$.

### *D. Discussion about the cost of the proposed algorithms*

In programmable switches, the best trade-off between TCAM memory usage (as per strategy proposed in [15]) and number of instructions in Arithmetic Logic Unit (ALU) should be explored: the processing time in the pipeline of programmable switches increases as the number of instructions in the ALU increases, but this relieves from the usage of TCAM memory. Avoiding the usage of TCAM memory implies that P4Log and P4Exp do not require any two-way interaction with the controller. In fact, when TCAM is used, the controller needs to populate ternary match tables during the setup phase. A thorough investigation and comparison with the state of the art in terms of overall memory occupation, data/control plane interaction and execution time, is left as future work.

## VI. Evaluation of P4Entropy

We implemented P4Entropy in Python and simulated it for evaluation. We report results in this section.

### *A. Evaluation metrics and simulation settings*

*1) Testing flow trace:* We use 2018-passive CAIDA flow trace [28] for evaluation into 10 observation windows with a fixed number of packets equal to $2^{21}$ each. Fixing the number of packets in an observation window is needed to compare our approach with a state-of-the-art solution [6], named *SOTA_entropy* for the remainder of the section, which can only be applied to observation windows where the number of packets is fixed to a power of two.

*2) Metrics:* We consider *relative error* as metric. We call $\hat{H}$ the estimated traffic entropy in an observation window and $H$ its exact value. The relative error is defined as the average value of $\frac{|H-\hat{H}|}{H} \cdot 100\%$ in the 10 observation windows.

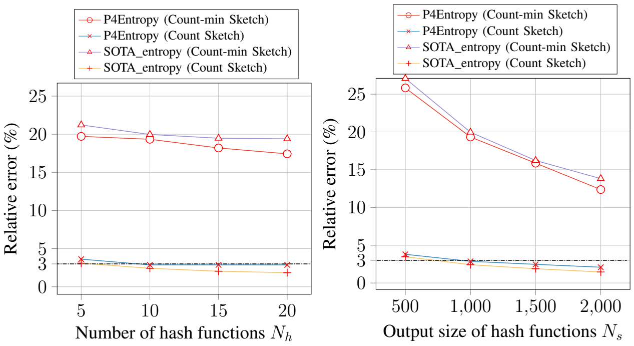
*3) Tuning parameters:* Unless otherwise specified, the default tuning parameters are set as per Table III.

### *B. Simulation results*

We simulate both our strategy and SOTA_entropy in the case that flow packet counts are estimated in the data plane by adopting either Count-min Sketch or Count Sketch (see Section II-D). We show how entropy estimation is affected while changing the size $N_h \times N_s$ of the sketch (Fig. 7). Fig. 7(a) shows the relative error in network traffic entropy estimation for the two strategies when $N_s$ is fixed and $N_h$ varies. It shows that the relative error slightly decreases as $N_h$ increases in all the cases. Moreover, P4Entropy and SOTA_entropy lead to similar relative error. It can be noted that, when adopting Count-min Sketch, both P4Entropy and SOTA_entropy have large relative error (around 20%) meaning that Count-min Sketch, with our settings, badly estimates flow packet counts $\bar{f}_i$ and both entropy estimation strategies result ineffective. Additionally, in this case, the relative error of SOTA_entropy is slightly higher than the one of P4Entropy, which is caused by the different ways how $Sum(f_i)$ is estimated. In SOTA_entropy, the Longest Prefix Match (LPM) lookup table for $F(f_i) = f_i \log_2 f_i - (f_i - 1)\log_2(f_i - 1)$ (see [6]) is sensitive to the large packet count ($f_i$) overestimation caused by Count-min Sketch. Conversely, P4Entropy needs to

TABLE III
DEFAULT PARAMETERS FOR P4ENTROPY

| | Alg | Parameter | Value |
|---|---|---|---|
| P4Entropy | P4Log | $N_{digits}$ | 3 |
| | | $N_{bits}$ | 4 |
| | P4Exp | $N_{terms}$ | 7 |
| Sketch size | | $N_h \times N_s$ | $10 \times 1000$ |



(a) Sensitivity to $N_h$ (b) Sensitivity to $N_s$

Fig. 7. Performance comparison of P4Entropy with an existing approach [6]

calculate $\log_2 f_i + \frac{1}{\ln 2}$ (see Eq. 1), which is less sensitive to large overestimations (i) due to the logarithm nature and (ii) because $\frac{1}{\ln 2}$ is a constant value. This effect does not happen while adopting Count-Sketch, since overestimations are much less frequent. In that case, P4Entropy leads to slightly worse results than SOTA_entropy because, unlike SOTA_entropy, it uses an approximation for the computation of the network traffic entropy (see Eq. 2).

Fig. 7(b) shows instead the impact of a variation of $N_s$ on relative error in entropy estimation. Results are similar to what shown in Fig. 7(a), but it can be noted that both strategies are more sensitive to a variation of $N_s$ than of $N_h$. In this case, when adopting Count Sketch, relative error is always close to 3%. Note that a relative error of 3% is the maximum possible value ensuring that accuracy of practical monitoring applications is not affected [1].

## VII. RELATED WORK

**Logarithmic and exponential function estimation in P4**: Since P4 language does not support logarithm and exponential function computation, many advanced algorithms leveraging on those operations (e.g., HyperLogLog for linear counting [29]) are not directly implementable using such domain-specific language. However, these advanced algorithms are useful for executing many network functionalities, such as congestion control [2], flow-cardinality estimation [15] and DDoS detection [6][30], so finding a way to support them is of paramount importance. Naveen et al. [15] have already successfully implemented estimation of logarithm and exponential function in P4, but their strategy requires the storage of appropriate pre-computed values in TCAM. It is shown that, to ensure a relative error in the estimation below 1%, they require around 0.5KB of TCAM memory occupation. This is something that P4Exp and P4Log algorithms do not need.

**Network traffic entropy estimation in data plane**: Many works can be found in literature dealing with network traffic entropy estimation partially performed in the switches' data plane. For example, SketchVisor [19], UnivMon [26] and Elastic Sketch [31] all envision some operations to be executed in the programmable data plane and send to the controller only summarized data. However, entropy estimation is executed at the controller due to the need of logarithm calculation. Our approach, instead, allows to compute the estimated entropy directly in the data plane, without any interaction with the controller. Additionally, Lapolli et al. [6] recently implemented network traffic entropy estimation in the data plane using the P4 language, with the aim of detecting DDoS attacks. Their approach is valid but they require the usage of TCAM, which is instead avoided by P4Entropy. Moreover, P4Entropy adopts a time-interval-based observation window, while [6] requires an observation window including a fixed power-of-two number of packets. Our approach is more beneficial since it allows a controller to synchronize the retrieval of estimated entropy among all deployed programmable switches to estimate traffic distribution on a *network-wide* scale [11], thus improving statistical relevance of monitored values.

## VIII. CONCLUSION

In this paper, we initially presented two algorithms, P4Log and P4Exp, for the estimation of logarithm and exponential function by only using P4-supported operations. The algorithms have been successfully implemented in P4 and can be used to enable the execution of monitoring tasks requiring the computation of such functions. Then, based on those algorithms, we proposed P4Entropy, a novel strategy allowing the estimation of network traffic entropy entirely in the data plane, which has been implemented in P4 as well.

We also evaluated all our proposed algorithms. We performed a sensitivity analysis of P4Log and P4Exp with the goal of finding the most appropriate values for different tuning parameters to guarantee a relative error between estimated and exact outputs below 1%. Unlike existing strategies in literature, our algorithms avoid the usage of TCAM. We also proved that P4Entropy has comparable accuracy to an existing approach but, as P4Log and P4Exp, does not require the usage of TCAM. Moreover, unlike the state of the art, P4Entropy does not need a fixed-packet observation window, being then more suitable when entropy estimation from multiple switches has to be delivered to the controller in a synchronous fashion.

As future work, we plan to conduct new experiments to evaluate *(i)* processing time, *(ii)* data/control plane interaction, and *(iii)* overall memory occupation of our proposed algorithms in the emulated environment. Furthermore, starting from our proposed algorithms, we plan to implement novel mechanisms and structures to enable advanced monitoring functionalities on a network-wide scale, such as entropy-based DDoS or traffic change detection.

## ACKNOWLEDGEMENT

The research leading to these results has received funding from the EC within the H2020 Research and Innovation program, Grant Agreement No. 856726 (GN4-3 project).

# References

[1] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 1, pp. 145-156, 2006.

[2] L. Jiang, D. Shah, J. Shin, and J. Walrand, "Distributed random access algorithm: scheduling and congestion control," *IEEE Transactions on Information Theory*, vol. 56, no. 12, pp. 6182–6207, 2010.

[3] K. Wu, L. Chen, S. Ye, and Y. Li, "A load balancing algorithm based on the variation trend of entropy in homogeneous cluster," *International journal of grid and distributed computing*, vol. 7, no. 2, pp. 11–20, 2014.

[4] Y. Gu, A. McCallum, and D. Towsley, "Detecting anomalies in network traffic using maximum entropy estimation," in *ACM SIGCOMM conference on Internet Measurement*, 2005.

[5] P. Bereziński, M. Szpyrka, B. Jasiul, and M. Mazur, "Network anomaly detection using parameterized entropy," in *IFIP International Conference on Computer Information Systems and Industrial Management*, 2015.

[6] A. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time DDoS attack detection to programmable data planes," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019.

[7] X. Ma and Y. Chen, "DDoS detection method based on chaos analysis of network traffic entropy," *IEEE Communications Letters*, vol. 18, no. 1, pp. 114–117, 2013.

[8] A. Wagner and B. Plattner, "Entropy based worm and anomaly detection in fast IP networks," in *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE)*, 2005.

[9] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2014.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM Computer Communication Review*, vol. 38, no. 2, 2008.

[11] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "Incremental deployment of programmable switches for network-wide heavy-hitter detection," in *IEEE Conference on Network Softwarization (NetSoft)*, 2019.

[12] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[13] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE mobile computing and communications review*, vol. 5, no. 1, pp. 3–55, 2001.

[14] "P4 behavioral model," https://github.com/p4lang/behavioral-model.

[15] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[16] H. S. Warren, "Hacker's delight," in *Pearson Education*, 2013.

[17] "The P4 language specification," https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[18] M. H. Quenouille, "A relation between the logarithmic, Poisson, and negative binomial series," *Biometrics*, vol. 5, no. 2, pp. 162–164, 1949.

[19] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[20] G. Cormode, "Count-min sketch," in *Springer Encyclopedia of Database Systems*, pp. 511-516, 2009.

[21] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Springer International Colloquium on Automata, Languages, and Programming (ICALP)*, 2002.

[22] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases*, 2011.

[23] "P4Log and P4Exp source code," https://github.com/DINGDAMU/P4Log_and_P4Exp.

[24] "Mininet," http://mininet.org/.

[25] "P4Entropy source code," https://github.com/DINGDAMU/P4Entropy.

[26] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[27] D. J. Struik, "The origin of L'Hopital's rule," *The Mathematics Teacher*, vol. 56, no. 4, pp. 257–260, 1963.

[28] "CAIDA UCSD anonymized Internet traces dataset," http://www.caida.org/data/passive/passive_dataset.xml.

[29] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *Discrete Mathematics and Theoretical Computer Science*, pp. 137-156, 2007.

[30] C. Wang, T. T. Miu, X. Luo, and J. Wang, "SkyShield: a sketch-based defense system against application layer DDoS attacks," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 559–573, 2018.

[31] 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 *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.