Secure Network Monitoring Using Programmable Data Planes

Fábio Pereira, Nuno Neves, Fernando M. V. Ramos
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
fpereira@lasige.di.fc.ul.pt, nuno@di.fc.ul.pt, fvramos@ciencias.ulisboa.pt

***Abstract*—The accuracy provided by traditional sampling-based monitoring approaches, such as NetFlow, is increasingly being considered insufficient to meet the requirements of today's networks. By summarizing all traffic for specific statistics of interest, sketch-based alternatives have been shown to achieve higher levels of accuracy for the same cost. Existing switches, however, lack the necessary capability to perform the sort of processing required by this approach. The emergence of programmable switches and the processing they enable in the data plane has recently led sketch-based solutions to be made possible in switching hardware.**

One limitation of existing solutions is that they lack security. At the scale of the datacenter networks that power cloud computing, this limitation becomes a serious concern. For instance, there is evidence of security incidents perpetrated by malicious insiders inside cloud infrastructures. By compromising the monitoring algorithm, such an attacker can render the monitoring process useless, leading to undesirable actions (such as routing sensitive traffic to disallowed locations). In this paper we propose, for the first time, a secure sketch-based monitoring solution that can run in programmable switches. Our algorithm – a secure version of the well-known count-min sketch – was implemented in P4, a programming language for switches. The evaluation of our solution demonstrates the performance penalty introduced by security to be negligible.

I. Introduction

Monitoring is essential for correct network operation. Ideally, for complete accuracy, the monitoring task should store all transmitted packets for subsequent analysis. In practice, however, this technique would lead to storage and processing scalability issues. Fortunately, exact results are usually not necessary, and a high quality approximation is enough. This fact suggests the use of probabilistic algorithms, that use smaller amounts of memory and require less computation to achieve the desired goals.

To avoid the storage and processing of all packets, traffic data can be reduced by *sampling*, with only a subset of the traffic being captured. This is the approach followed by NetFlow and sFlow, the most widely-used techniques. To be scalable, however, the sampling frequency of these solutions is kept at low levels (a common figure is 1:1000). This reduces the accuracy to a level that precludes its use for many of the advanced monitoring capabilities required in today's large scale networks that enable cloud computing.

An alternative approach is sketching [5]. With this approach the monitoring task takes into account *all* packets, instead of only a subset. To maintain memory and processing at acceptable levels, these algorithms summarize the network data streams in the data plane (by employing hashing, counting, and filtering techniques). These solutions have been shown to offer an interesting trade-off between the accuracy achieved and the memory used, outpacing the alternative for various monitoring tasks. Existing switches, however, lack the necessary capability to enable this approach.

The emergence of programmable switches has given operators the opportunity to run complex processing in the data plane, radically changing the state of affairs. Recent proposals [6], [7] have shown the feasibility of sketch-based solutions in real hardware data planes. One limitation of existing solutions is that they lack security properties. Indeed, if the monitoring algorithm itself is not secure, its results may be corrupted. In the worst-case scenario, the network administrator does not notice the results are corrupted, and takes improper actions. For instance, an attacker may persuade the monitoring system to route sensitive traffic to a location he or she controls. Unfortunately, there is evidence that the problem is real. A recent report mentions malicious insiders as one of the top threats in cloud computing [8], and alarming instances of this problem have been show to occur [9]. The security limitation of current approaches is therefore already a serious concern.

To the best of our knowledge, no attempt has hitherto been made to address the security of sketch-based algorithms. Our work starts filling that gap. We propose a secure version of a sketch-based algorithm – Count-Min – that enables secure traffic monitoring. Our solution addresses several technical challenges, many of which arise from the constraints imposed by real switches. These include the use of cryptographic hash functions (not supported in existing switches), avoiding loops (not directly available as they would limit throughput), and techniques for secret key renewal. We prototyped our solution in P4 [2], a programming language for network switches. Our evaluation using the public-domain behavioral P4 switch model [1] demonstrates that securing the sketching algorithm introduces a negligible performance penalty.

II. Background and Related Work

Sketching algorithms. In contrast with sample-based techniques, this type of monitoring solution processes *every* packet, performing a summarization (mainly by hashing and counting) for a specific statistic of interest. Importantly, the algorithms are designed with provable accuracy-memory tradeoffs. In this paper we focus on one of the most well-

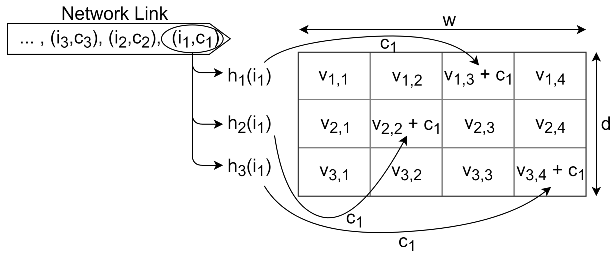

Fig. 1. Count-Min update operation with a data structure with width w = 4 and depth d = 3.

known sketching algorithms: count-min. This sketch solves the *Count Tracking* problem, where the goal is to find the frequency of each item in a stream with a large number of items [4]. In the network monitoring context, the approach can be used for example to count the number of packets transmitted from a specific source or to count bytes sent from that source.

Like in other sketch-based algorithms, the Count-Min algorithm requires a data structure to store the information about the packet stream and provides two basic operations:
– *Data Structure:* is a two-dimensional array of counters with w columns and d rows, both fixed at the time of creation (illustrated in Figure 1). These values, w and d, are chosen based on the desired accuracy of the estimates. The counters are initialized with zero. The algorithm uses d hash functions from a pairwise-independent family that produce values uniformly at random. At update time, each of these functions maps the *item description* onto the range $\{1, 2, \ldots, w\}$.
– *Update(i,c) operation:* when a new item i arrives, then for each row the corresponding hash function is applied to i to determine the column that needs to be modified. Next, a value c (either positive or negative) is added to the target counter.
– *Estimate(i) operation:* to estimate the frequency of an item i, it is necessary to find the values stored in each row for that item by applying the corresponding hash function. The estimate is the smallest value stored in all counters.

Accuracy: sketching a stream normally causes some loss of accuracy. To minimize this loss, the sketch dimensions should be as high as the target device allows. This way, the collision probability is lower and, as a consequence, the average accuracy of the estimates will be higher. Another factor contributing to the accuracy is the duration of the monitoring cycles. Whenever the counters are restarted, the accuracy of the sketch is perfect, starting to decrease after the occurrence of collisions.

It has been shown, that if N is the sum of the values of all the counters in a row of a sketch, the frequency of an item i returned by the algorithm is at most $\frac{2}{w}$ of N more than its true frequency, with a probability of at least $1 - \frac{1}{2}^{d}$.

Programmable data planes. Common switching chips are fixed-function. They run a fixed set of protocols, defined at manufacturing time, and the sort of packet processing available is therefore restricted. Emerging programmable switching hardware [3] gives an unprecedented level of flexibility to packet processing. To program this new generation of switches an open-source language, P4 [2], has been proposed, already counting with significant support from the industry. A P4 program enables the definition of packet headers and its parsers (how packet headers should be extracted), of the set of match-action tables and the list of available actions (how packets can be manipulated), and finally of the entire control program (the sequence of operations that determine the order in which the match-action tables are applied to each packet).

Leveraging on these advances, recent work has proposed solutions that enable, for the first time, sketch-based algorithms to run in hardware switches. Liu et al. [7] have proposed UnivMon, a framework that allows universal streaming: a single universal sketch that is shown to be provably accurate for estimating a large class of functions. In [6], V. Sivaraman et al. propose a heavy-hitter detector that works entirely in the data plane. These solutions were both prototyped in P4. Contrary to our proposal, none of these works considers security in their design.

III. Secure Count-Min Sketch

Today's network links operate at very high speeds, decreasing the time that can be spent processing each packet. This constraint has led to monitoring approaches that completely neglect security in favor of ones that minimize the time and space requirements. In some controlled environments this lack of security might be acceptable because threats are limited, but in general it is difficult to assume that no attacks will ever occur. In addition, monitoring activities are often used in the context of network defense applications, such anomaly detection and intrusion prevention. Therefore, if the monitoring algorithms are insecure then their results may not be trustworthy, what makes their activities worthless or, in a worse case, counter-productive — since corrupted results could lead the network administrator to take inappropriate actions. Therefore, choosing a secure version of a monitoring is crucial to ensure that the decisions are always adequate.

A. Attack Model

We assume an adversary that might be anywhere inside the network but that has not compromised the device where the monitoring solution is deployed. All details about the implemented algorithms are known to the adversary, and therefore he may be able to perform the following actions.

Eavesdrop: If the adversary is placed right before the monitoring device, he can observe exactly the same traffic. Since he knows all the implementation details of the monitoring algorithm, it becomes possible to predict all the actions that will be taken.

Drop packets: Assuming that the adversary possesses the same knowledge as the legitimate monitoring task, he can drop some packets in order not to trigger a specific event by the algorithm, which could uncover an attack being executed in background. The dropped packets may be chosen in a way that simulates the usual losses of the network, without any suspicious activity.

Modify packets: If an adversary can capture, modify and then replay the packets being transmitted without being noticed, he will be able to corrupt a monitoring algorithm that does not ensure the authenticity of the monitored traffic. The adversary can, for example, modify the packets in such a way that they will collide when the algorithm's hash functions are applied to them. This would cause counters to be incorrect. Another example attack is the overflow of counters before the monitoring entity reads their values. This may be specially destructive if some action is programmed to be taken when a counter is close to its limit. For example, right before counters overflow, their values may be collected and written to a slower memory. Since in normal situations each counter overflows at a different time, the algorithm may not be designed to handle situations in which there are many counters overflowing at the same time.

Generate traffic: Assuming that the monitoring task is keeping track of the frequency of each source IP address, the adversary can spoof his IP address to one that, by applying the algorithm's hash function, will collide with an IP address of a legitimate user. By repeating this action, the adversary can trick the monitoring task into thinking that a specific legitimate user is generating more traffic load than he truly is.

B. Sketching in a Secure Way

Our objective is to take the Count-Min sketch algorithm and modify its operation in such a way that it is no longer vulnerable to attacks, but without compromising its accuracy, performance, and simplicity of design.

Many of the problems that were identified can actually be prevented if the attacker is no longer capable of predicting the behavior of the algorithm. In particular, if he is unable to guess which entries in the Count-Min data structure are modified with the arrival of a packet, then he cannot emulate the algorithm behavior just by observing the arriving traffic.

Therefore, an effective way to achieve this goal is to substitute the original hash functions by a fast cryptographic hash function that receives as input also a strong key (128-bits). Since the key is unknown to the adversary, it becomes extremely hard to brute-force in an attempt to create for instance hash collisions. Since network monitoring is often required to be continuously active, we need to provide the possibility to change this key at runtime (i.e. without having to restart the switch). The periodicity of the key change is decided by the network administrator, who should consider the accuracy guarantees of the Count-Min sketch (see II). The sum of a line of the sketch could be a good metric to decide if it is necessary to renew the key as accuracy benefits from keeping this value low. In addition, the existence of counters that are about to reach their maximum value should also trigger a key change.

However, it is not possible to exchange the key and keep using the same data structure because two equal items would be mapped to different positions. Lets call *monitoring period* to the interval of time during which the same key is used by a switch. The key should be renewed at the end of each monitoring period and the data structure copied to a different memory before its clean up. All the estimate operations access not only the data structure being used by the switch but also the data structures previously stored.

It should be noted that estimations based on more than one data structure will not have the same accuracy guarantees as the original Count-Min algorithm, based on a single data structure. For each stored data structure, a estimation for the given item is done and the final estimation returned by the algorithm is the sum of the individual estimations. This means that the final estimation will have an error of at most the error of the Count-Min algorithm multiplied by the number of data structures the estimation is based on. However, since item values are typically obtained periodically for a restricted period, this means that the administrator only needs to keep the stored data structures that are still needed for the measurements. The older ones can be deleted, which ensures that the accuracy is only affected in a very limited way.

Algorithm 1 Update Operation

```
1: // constants gathered during switch initialization
2: width, height, cSize = read("inputFile")
3: /* Stateful memories that persist across packets */
4: lastRow = width * (height - 1)
5: // array of counters with cSize bits each
6: c = Array[width * height * cSize]
7: // allocate 128 bits to store the hash_function key
8: key_register = Register[128]
9: /* Executed to every packet that arrives */
10: procedure UPDATE(PACKET)
11:     // item can be any field of the packet header
12:     item = PACKET.src_ip
13:     // read the current version of the key
14:     key = read_register("key_register")
15:     targetRow = 0
16:     while targetRow <= lastRow do
17:         // set the input of the hash function
18:         hash_input = {item, key, targetRow}
19:         // get a column number
20:         targetColumn = hash(hash_input) % width
21:         // get counter of that column in the current row
22:         targetSlot = targetRow + targetColumn
23:         // ensure that overflows do not occur
24:         if c[targetSlot] < 2^cSize - 1 then
25:             c[targetSlot] = c[targetSlot] + 1
26:         end if
27:         targetRow = targetRow + width
28:     end while
29:     forward_packet(PACKET)
30: end procedure
```

1) The Algorithm: In order to facilitate the dynamic deployment of the monitoring algorithm in the network, we have designed it to run in programmable switches supporting the P4 language. Below, we present the main operations that have to be implemented:

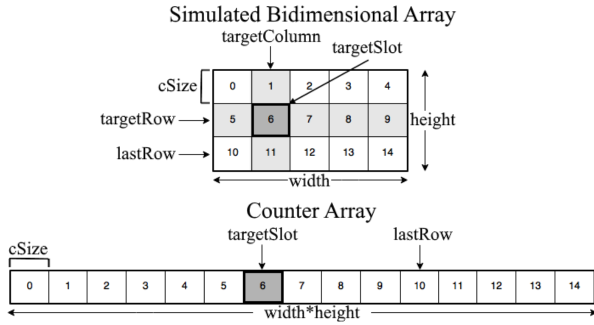

Fig. 2. Count-Min bidimensional array simulated into a linear array.

Update Operation: Algorithm 1 implements the operation that processes each arriving packet. The initial part of the algorithm defines the variables that must persist across packets (Lines 1-9). The main data structure of the algorithm is the array `c` of counters. Its dimensions are read from a configuration file. The `key_register` allocates memory to store the key. An external function accesses this memory to keep it updated with the current version of the key.

The second part of the algorithm is executed for every packet that arrives. `item` defines the information that is tracked by the sketch, which in the displayed code is the source IP address of the packet (Line 12). The current version of the key is read from the `key_register` memory so it can be used by the hash function (Line 14). Next, the algorithm loops through each row of the array to modify a counter. The hash function uses as input the key, the `item`, and the row number to determine the column/counter that should be updated (Lines 17-22). Figure 2 illustrates how an access to the `c` array corresponds to an access of a Count-Min data structure. Then, the counter is incremented (Lines 24-26). It is possible to observe that the algorithm precludes the wrap around of counters (Line 24) to prevent certain forms of attack, such as making a heavy hitter source look like a normal sender (e.g., to avoid DoS attack detection).

Estimate Operation: Algorithm 2 produces an estimate for the value being monitored. It is implemented in two procedures, the first that loops through all `c` data structures that exist (current and past) to get the global estimate (Lines 1-15). The second provides an estimate based on a single data structure by searching for the minimum value of all counters associated with the `item` (one per row) (Lines 17-38). For example, if the program defines `item` as the source IP address, then the algorithm returns an estimation for the number of packets that were received from `item`.

The `ESTIMATE` procedure starts by fetching the data structures that were stored previously whenever there was a key update (and which were not deleted by the administrator) (Line 5). Then it gets an estimate from each one of them (Lines 6-10), and also from the one being currently utilized by the switch (Lines 12-13). The sum of all estimations is the value returned by the procedure.

The `ESKETCH` procedure calculates the target counter to be read (Lines 18-25), one per row of the data structure, and then returns the minimum value that was found (Line 34).

Algorithm 2 Estimate Operation

```
1:  /* Estimates the frequency of item */
2:  procedure ESTIMATE(ITEM)
3:      est = 0
4:      //file with past arrays and respective keys
5:      old_sketches_file = "old_sketches"
6:      for each element in old_sketches_file do
7:          oldSketch = element.get_sketch()
8:          key = element.get_key()
9:          est = est+ ESKETCH(oldSketch, key, ITEM)
10:     end for
11:     //get the key currently being used by the switch
12:     key = read_register("key_register")
13:     est = est+ ESKETCH(NULL, key, ITEM)
14:     return est
15: end procedure
16:
17: procedure ESKETCH(SKETCH, KEY, ITEM)
18:     width, height, cSize = read("inputFile")
19:     targetRow = 0
20:     lastRow = width × (height − 1)
21:     result = +∞
22:     while targetRow <= lastRow do
23:         hash_input = <ITEM, KEY, targetRow>
24:         targetColumn = hash(hash_input) % width
25:         targetSlot = targetRow + targetColumn
26:         if SKETCH == NULL then
27:             // read from data structure kept in the switch
28:             rowRes = counter_read("c", targetSlot)
29:         else
30:             // read from a stored sketch
31:             rowRes =SKETCH[targetSlot]
32:         end if
33:         // calculate the minimum between obtained values
34:         result = minimum(result, rowRes)
35:         targetRow = targetRow + width
36:     end while
37:     return result
38: end procedure
```

IV. Implementation

The implementation of the algorithm in P4 presented a few challenges, not only because of limitations of the language but also due to the constraints imposed by the interface between a P4 program and the software switch. This section briefly explains how the main obstacles were overcome.

One-dimensional array: The main data structure of the algorithm had to be linearized to an array with a single dimension (instead of a two-dimensional array like in the original algorithm's description). There are two reasons for this modification: (i) P4 does not support multi-dimensional arrays; and (ii) the size of the array has to be specified at

start time (no dynamic allocation). Consequently, to define the desired data structure's size, the user writes a configuration file with the width, height and number of bits in each entry. To initiate monitoring, a script is run that starts by reading that configuration file and, based on it, calculates a set of values that are written to a constants file. Those values are then used inside the P4 program through the preprocessing directive `#include`, which makes the content of the constants file available to the P4 code.

The P4 code bellow shows the definition of the main data structure, which has `NUMBER_OF_INSTANCES` entries (equal to $width * height$), each one with `SLOT_SIZE` bits. The attribute `saturating` prevents counters to wrap around by stopping to count if they reach their maximum value (according to P4 version 1.0.3).

```
counter counters{
  type: packets;
  instance_count: NUMBER_OF_INSTANCES;
  min_width: SLOT_SIZE;
  saturating; //prevents overflows
}
```

Repeat actions: P4 does not support loops. Therefore, a workaround had to used: (i) the loop is unrolled, creating an if statement per iteration; (ii) the if condition stops processing (becomes false) when a counter reaches a previously defined maximum value. Below, it is exemplified a loop through the lines of the data structure, which could be executed up two times (depending on the value of `LAST_ITERATION`).

```
if(c_metadata.target_row <= LAST_ITERATION)
    apply(update_table1);
if(c_metadata.target_row <= LAST_ITERATION)
    apply(update_table2);
```

Since P4 does not allow an action to be applied to the same table more than once, we had to create distinct tables per iteration. To ensure the desired effect, all these tables are associated with identical actions and memory locations, as exemplified in the next code listing.

```
table update_table1{
    actions{update_row;}
    size: 1;
}
table update_table2{
    actions{update_row;}
    size: 1;
}
```

Hash function: To implement the hash, we used the "simple_switch" target from the behavioral-model [1]. However, it restricts the exchange of data between the switch and the P4 program to 64 bits, which is not enough to hold a MD5 hash function output (128 bits). To overcome this, we divided the hash computation in two calls, where one calculates the actual MD5 function and returns the 64 most-significant bits of the result and the other simply returns the 64 less-significant bits (of that same result). The following code shows how the 64 bits returned values are aggregated into a 128 bit field in P4.

```
modify_field_with_hash_based_offset(
    c_metadata.full_hash, 0, hash_p1, MAX);
modify_field_with_hash_based_offset(
    c_metadata.second_part, 0, hash_p2, MAX);

shift_left(c_metadata.full_hash,
    c_metadata.full_hash, 64);
bit_or(c_metadata.full_hash,
    c_metadata.full_hash,
    c_metadata.second_part);
```

Constant `MAX` has value of 2^{64}, which is required by the P4 function `modify_field_with_hash_based_offset` to apply a modulo operation to the result of the hash function.

V. Evaluation

We carried out two sets of experiments to evaluate the performance of our algorithm. The first group measured the latency and throughput of the secure count-min sketch, while the second set tested the error estimations in several settings.

The testbed was composed of three machines, where one emulates a P4 switch inside a mininet instance connecting the other two. The switch machine was an Intel(R) Xeon(R) CPU E5-2407 v2 @2.40GHz with 64GB of RAM, and the other hosts were standard Intel PCs. The virtual switch implemented the second version of P4, known as behavioral-model (bmv2) [1]. It was loaded with our P4 algorithm to monitor and forward the received traffic.

A. Performance of traffic forwarding

The developed solution was compared against two other P4 programs: (i) the original Count-Min algorithm was used as baseline; and, (ii) we employed a program that simply forwards the traffic to understand the cost of monitoring. As explained previously, the size of the data structure of the count-min sketch allows to trade accuracy for overheads, as for each additional line in the data structure there should be a performance degradation due to the computation of an extra hash function. Therefore, we provide experimental results for different number of lines in the data structure.

1) Latency: To measure the delay introduced by the switch, we calculated the average round trip time (RTT) of 10000 pings between the two end hosts. Figure 3a shows the observed average latency and standard deviation for the three P4 programs. As expected, the forwarding program shows a constant latency of around 500 microseconds. The secure version of Count-Min performed worse than the original algorithm, with an average overhead of around 10%. The difference between them was approximately 40 microseconds for a sketch with one line, and around 160 microseconds for a sketch with 20 lines. This is an interesting result because it demonstrates that with a relatively small rise in the overheads, it is possible to offer increasingly low error probabilities (see II). Of course, if there was hardware support for P4, one should see a significant decrease on these values.

2) Throughput: The throughput was measured with *iPerf* between the two nodes. In order to obtain the maximum throughput, the traffic rate of iPerf was increased until the

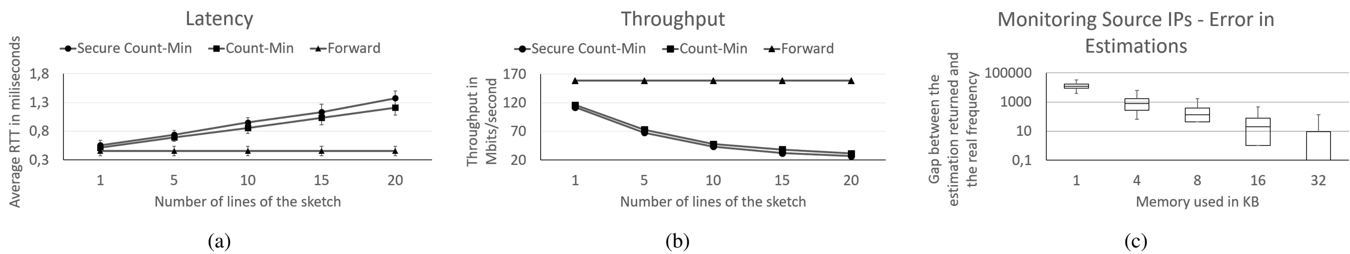

Fig. 3. (a) Latency between the two hosts; (b) Maximum throughput; (c) Estimation errors when monitoring by source IP address

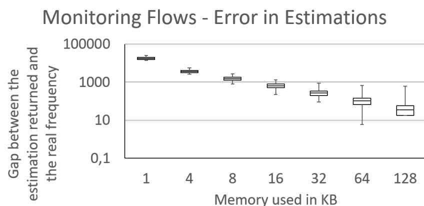

Fig. 4. Estimation errors when monitoring by flow identifier

network started to drop packets (i.e., loss rate > 0). Although each experiment was repeated 20 times, the calculated standard deviation was not big enough to be observable in Figure 3b.

The forward-only solution got the best results, achieving a throughput of around 159 Mbits/second. The performance cost imposed by adding security to Count-Min was on average about 5 Mbits/second. Our secure Count-Min algorithm achieved a throughput of 112 Mbits/second with a data structure with 1 line. With 10 lines, the throughput drop was to 44 Mbits/second and to 27 Mbits/second with 20 lines.

B. Observed errors in estimations

The error associated with the estimations returned by our secure sketch was also evaluated. We used a five minute trace of IPv4 traffic captured on busy private network's access point to the Internet. We injected this set of well known packets in our network so the switch could process them all. Then, for each distinct monitored item (*source IP* or *flow identifier*), we queried the sketch for that item's estimated frequency. The difference between the estimated frequency and its true frequency is the estimation's error of that item.

The goal was to compare how different data structure dimensions would affect the errors observed in estimations of two monitoring conditions — count packets by sender (i.e., source IP address) and by flow (i.e., 5-tuple with source/destination ports and IPs for TPC connections). Since there are many more distinct flows (22310) than source IP addresses (1845), we expect to see larger errors when monitoring by flow for a given memory size. The number of lines of the data structure was fixed to ten, since it would only affect the probability of the error and not its extent. To test different memory usages, we used data structures with increasing numbers of columns, starting from 25 (1 KB).

1) Source IP: In this case the monitored item was the source IP address of 791179 packets. Calculating the estimations error of all items allowed us to find the minimum and maximum error, and the percentiles 10, 50 (median) and 90 for each experiment, as displayed in Figure 3c.

It was observed that the error decreases as the data structure's size increases. For a data structure with width 25 (1 KB if its height is 10 and each counter occupies 32 bits) the median of the errors was around 12000, which may not be tolerable. However, for a data structure with just 400 columns (16 KB required), the sketch could already achieve relatively small errors, with a median of around 20. Finally, by using 32 KB of memory, the median of the errors was 0.

2) TCP Flows: We also monitored the traffic by flows, where the total number of relevant packets sent through the monitoring device was 633746 (corresponding to all TCP packets). Figure 4 shows that a data structure with 25 columns (1 KB) leads to a median of errors around 18000. As the memory used increases, the error in estimations decreases, with a median of 646 when 16 KB of memory was used. We also tested our solution using 128 KB of memory, where the median of the errors was only 36.

VI. Conclusion

This paper proposed a secure sketch-based monitoring algorithm. We secured the Count-Min sketch and adapted it to a network monitoring context. Our prototype was implemented in P4, leveraging from programmable data planes that have recently become available. The experiments show that making a sketch secure does not introduce relevant performance penalties in terms of latency and throughput.

References

[1] P4 software switch (behavioral model) https://github.com/p4lang/behavioral-model. Accessed: June 2017

[2] P. Bosshart et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 2014.

[3] P. Bosshart et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM 2013*

[4] G. Cormod et al. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.

[5] A. Gilbert et al. Quicksand: Quick summary and analysis of network data. Technical report, 2001

[6] V. Sivaraman et al. Heavy-hitter detection entirely in the data plane. In *SOSR 2017*

[7] Z. Liu et al. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM 2016*.

[8] Cloud Security Alliance, The notorious nine: Cloud computing top threats in 2013.

[9] M. Kandias et al, The insider threat in cloud computing, in *Critical Information Infrastructure Security*, ser. LNCS. Springer Berlin Heidelberg, 2013, vol. 6983, pp. 93–103.