A Superspreader Detection Algorithm in the Dataplane

Rui Yang
r3yang@eng.ucsd.edu
University of California, San Diego
San Diego, California

Weifeng Hu
weh031@eng.ucsd.edu
University of California, San Diego
San Diego, California

ABSTRACT

A superspreader is defined as a host that contacts with at least a given number of unique destinations in a short period of time. Superspreaders are responsible for top network attacks such as worm propagation and Distributed Denial of Services. Existing work on superspreader detection includes using data plane as a mean to collect statistics and using control plane to run analysis. Although such mechanism has shown promising result, the overhead of communication limits the ability for a controller to react in a timely fashion. In this paper we present a superspreader detection algorithm based on a multi-staged hash table implemented entirely in the data plane and prototype it in P4. We evaluate our algorithm on the packet traces of an ISP backbone link. Experiments show that our algorithm achieves a more than 95% precision and 79% recall, at the same time, outperforms the Elastic Trie Algorithm in Seek and Push[6], which is the state-of-art superspreader detection algorithm in the data plane.

KEYWORDS

superspreader, anomaly detection, algorithms, data plane

ACM Reference Format:
Rui Yang and Weifeng Hu. 2019. A Superspreader Detection Algorithm in the Dataplane. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software defined network (SDN) has gained popularity by decoupling the control logic from forwarding devices. However, it also introduces potential vulnerabilities and threats [5]. Among the threats exposed by SDN, the Distributed Denial of Services (DDoS) attack and the worm attack have been some of the most serious problems towards the network. In both attacks, one host would make an unusually high number of connections to distinct destinations within a short period of time. We call such a host a **superspreader**. With the purpose of responding immediately to these attacks, it is desired to run superspreader detection at switches all the time. Figure 1 shows a toy example of a superspreader with five unique connections. Each node represents a host and the edge represents a connection. In this exmaple, if we define the threshold of becoming a superspreader is to have connections with at least 5 hosts, the red node will become a superspreader.

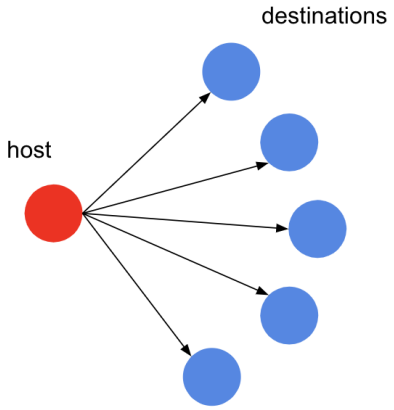

Figure 1: A toy example of a superspreader

The existing superspreader detection mechanism includes using data plane as a mean to collect statistics and using control plane to run analysis [7, 8, 10]. In simplicity, they use an application running on top of the controller to poll counters from switches. Even though such methods are able to obtain promising results, the communication overhead among application, control plane and data plane reduces the ability for the controller to react in a timely fashion. As superspreader attacks happen in a very short time scale. Some popular exisiting approaches including sampling packets in the form of NetFlow. However, with today's throughput in the network, even sampling one packet from 1000 packets can result in a large transmission overhead. Sketching packets with hashing and counting is another popular method performed in the control plane but it comes with a large memory usage overhead.

Emerging programmable switches enable us to do a lot more than simply sampling, counting and hashing. It makes performing complicated streaming algorithms directly on the data plane a reality without hurting the throughput of packets in the network. As a result, a number of proposals have been proposed to extend the functionality on data plane with P4 programming language [4]. Performing superspreader detection in the data plane allows switches to react immediately when finding anomaly behavior. For example, the switches can drop packets with source IP address matching the table of superspreaders. We can also have the option to report findings to control plane only when needed.

However, such implementation entirely in the data plane has some challenges. Programmable switches have limited memory resources compared to applications running on top of the controller.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The functionality therefore needs to be memory efficient. For example, space saving algorithm is intended to be memory efficient. Moreover, programmable switches is a restrictive programming model. Figure 2 shows a common PISA programmable switches as described in [4]. It includes a parser to parse the headers, a series of pipelines for match-actions and a deparser. The packet that goes through the deparser can either be forwarded to next hop or recirculate back to the parser. This kind of programmable model only enables a limited number of access to memory storing at each stage. More specifically, only one read-write at each stage can be performed at most time.

There are already some attempts of moving calculation and detection to the data plane. For example, currently heavy-hitter detection has been fully implemented in the data plane [2, 9]. Heavy-hitter is simply the elephant flow in the network, which another significant detecting object. Heavy hitter detection is to identify source IPs that contribute to most flows in a network. Jan Kucera et al. [6] has provided a Elastic Trie structure in the data plane that can detect heavy-hitters, hierarchical heavy-hitters and superspreaders. To the best of our knowledge, there is no data plane implementation that specifically targets superspreader detection.

In this paper, we proposed a superspreader algorithm in P4 and test it on the public-domain behavioral switch model. We evaluate our algorithm with traces in packet level of a ISP backbone link in Chicago (from CAIDA) and show that our algorithm can provide high accuracy with limited usage of memory space. In the backbone link traces, our algorithm manage to achieve more than 95% precision and 79% recall when reporting different thresholds of superspreaders and outperforms the seek and push algorithm[6], which is the state-of-art algorithm in the data plane. It also achieves less than 5% false negative rate when reporting 100 superspreaders with merely 440 counter in the multi-staged hash table.

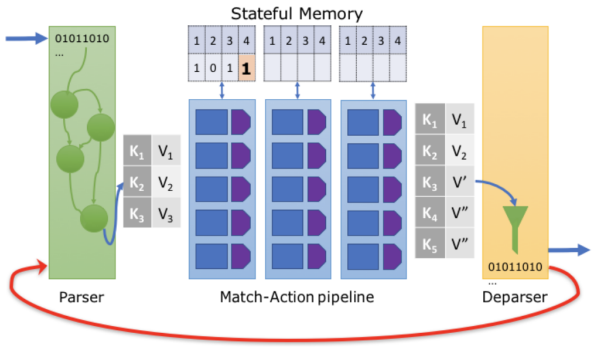

Figure 2: An example of PISA programmable switch

The rest of the paper is structured as follow. Section 2 talks about the related work in superspreader detection. Section 3 discusses the challenges we face when implement our algorithm in the data plane. Section 4 describes our superspreader detection algorithm in detail. Section 5 discusses the evaluation we performed. Section 6 talks about the conclusion and future work.

2 RELATED WORK

The existing implementation of superspreader detection leverages on collecting data from data plane and running analysis on control plane. Zaoxin Liu et al. [7] presents Univmon, which uses an application-agnostic data plane monitoring primitive, runs different estimation algorithms in the control plane and uses the statistics from the data plane to compute application-level metrics. They were able to achieve an error rate of less than 1% in DDoS detection, which is similar to superspreader detection.

Minlan Yu et al. [10] presents OpenSketch, which provides a simple three-stage pipeline (hashing, filtering, and counting) in the data plane and provides a measurement library that automatically configures the pipeline and allocates resources for different measurement tasks in control plane. They achieved 0.9% false positive rate and 0.3 % false negative rate in superspreader detection.

Qun Huang et al. [8] presents SketchVisor, which deploys a centralized control plane to merge the local measurement results from all software switches to provide accurate network-wide measurement. They achieved 90% recall and 84% precision in superspreader detection.

Jan Kucera et al. [6] is the only superspreader detection implementation in data plane to the best of our knowledge. It implements a Elastic Trie data structure in switches. The algorithm is intended to identify heavy hitters, hierarchical heavy hitters and superspreaders. The achieved a 60% recall and 85% precision in superspreader detection experiment. We redeem this work as the state of the art for superspreader detection.

As mentioned in the introduction section, we are not aware of any existing data plane only implementation specifically for superspreader detection. We believe that our algorithm is the first to address such issue.

3 DESIRED PROPERTIES & CHALLENGES

Superspreader detection can be achieved by keeping a list of unique destinations contacted for every source IP address. We can then sort the source IPs based on the length of its list. However, there are some challenges to implement such scheme entirely in the data plane.

3.1 Memory Efficiency

In order to implement superspreader detection entirely in the data plane, the algorithm needs to be able to run with limited memory resources in a programmable switch. In particular, keeping a list of unique destinations for every source IPs in the switch is too expensive to be achieved. As a result, we take advantage of Bloom filter [3]. Bloom filter is a memory efficient probabilistic data structure commonly used to test for set membership. We use the Bloom filter to keep track of the distinct destinations contacted by a source IP. Moreover, the Bloom filter can be easily implemented in P4 as a bit array placed in a register and a set of k hash functions. We study the effect of Bloom filter size on the performance of our algorithm in the evaluation section.

Another challenge in memory efficiency with implementing superspreader detection entirely in the data plane is that keeping track for every source IP requires an unbounded amount of memory. Considering superspreaders are a very small portion of sources,

it is ideal to use memory space proportional to the number of superspreaders instead of the total amount of sources. We will address this challenges in our algorithm section.

3.2 Accuracy

Considering the constraint of memory and restrictive model of p4, some popular streaming algorithms are hard to be implemented in the data plane without and discount, which makes the accuracy of data plane algorithms a concern since a trivial detection model could result in a lot of chaos to the operation of network. It is possible that data plane algorithms can not achieve same performance as the best models in the control plane, however, we aim to provide an accuracy that is good enough and matches the state of the art [6]. Our advantage is that we reduce the communication overhead between data plane and control plane, making the reaction time of switches much shorter.

4 ALGORITHM

In this section we will discuss how we come up with our superspreader detection algorithm in the data plane. Since there are no existing work on solely superspreader detection in the data plane, we take our inspiration from two heavy-hitter detection algorithms entirely in the data plane.

4.1 Inspiration

In order to develop a memory-efficient superspreader detection algorithm for programmable switches, we take our inspiration from the existing heavy hitter detection algorithm in the data plane [2, 9]. Both methods use a hash-based multi-stage table to count the number of flows. The size of the table is fixed to prevent it from growing unbounded. Each table entry is a key-value pair. When a packet comes in, at each stage we hash the key, which is part of information in the packet (e.g. source IP), to an index in the table. If the table entry on that index is empty, we simply insert the new entry. If the key on that table entry matches the information in the new packet, we simply increment the counter. There is a third situation where the table entry has its own recorded source IP which does not match the source Ip of the incoming packet. The differences between those two algorithms is how they handle such a hash collision, which is defined as two different keys mapped to the same index in the table. As shown in Figure 3, Hashpipe uses an aggressive methods at the beginning. To summarize, when a packet comes in, hashpipe always inserts that packet at the first stage. It then carries the information of replaced entry as metadata to the next stage. At next stage, if the counter in the metadata is larger than that in the table entry. The process is repeated until the last stage.

PRECISION algorithm [2] takes a different and comparatively conservative approach. Instead of inserting at first stage directly, PRECISION goes through all the stages and keeps track of a rolling minimum. The problem is that even though we know the entry of minimum count to replace, we cannot replace it directly because we have already left that stage. As a result, PRECISION recirculates the copy of the packet with a probability in order not to interfering the packets' order in the link. If the packet is recirculated, it will replace the minimum entry and the copied packet will be dropped.

These two algorithms both achieved satisfactory performance on the heavy hitter problem. Considering the similarity between heavy hitters and superspreaders, it is to our instinct to naively transplant them to do the superspreader detection. However, they both perform poorly in our experiments. This could be the following reasons. Hashipe always inserts the incoming packets at the first stage, which is simply too aggressive. In a series of same traces, it is obviously that the number of distinct flows (5 tuple) is larger than that of distinct source IPs, which means there are simply more different incoming flows than source IPs with a hash table of the same size. Under such circumstances, if an incoming mice flow is inserted into a entry of the table, it gets higher chance to be evicted later than if it is a source IP because of its small counter value and more hash collisions.

On the other hand, PRECISION, can avoid the problem of inserting in the first stage, but uses an algorithm not working for superspreader detection when calculating the possibility of recircualtion. Precision simply calculate the reciprocal of the smallest counter value the incoming packet experiences and multiple it with a self-defined probability, which does not fit the distribution of our data. In the superspreader detection problem, actual number of superspreader is very low and most source IP only contacts one destination in our time period and only a very small portion of source IP contacts more than 2 destinations. To be precise, with a counter value of one, the probability of recirculation should become much higher than when counter value is 2, 3 or even higher. Dealing with these situations evenly can result into poor prediction. Moreover, PRECISION calculates a recircualtion rate with every incoming packet. Even it is probabilistic, an decrease in the throughput of the switch can still be noticed.

Moreover, these two algorithms have a shared drawback which is the duplicated entry problem. Because of the insertion at first stage in Hashpipe and the consistency problem in PRECISION, there can be entries with same flow ids in the table in both these two algorithms. Now considering the origin of these two algorithms – the space saving algorithm. The key idea of space saving algorithm is to use k entries to detect k objects which makes even one duplicate entry in the table result in an obvious decrease in accuracy.

In the following section, we are to talk about the algorithm we have designed to deal with superspreader detection. Our algorithm solves the duplicate entry problem utilizes an policy with consideration of the distribution of superspreaders and achieves less than 5% false negative rate.

4.2 Table Entry

Different from the heavy-hitter detection algorithm, which only keeps the count of flows in a table, we need to keep track of a list of unique destinations contacted by a source IP and only increment the counter if the destination does not appear in the list. As a result, our table entry is shown in Figure 4. In this table entry the source IP is the key. When a packet comes in, we hash the source IP in that packet, and then we check if the destination IP in that packet is present in the Bloom filter. If not, we can increment the counter and update the Bloom filter. Otherwise, no action is performed.

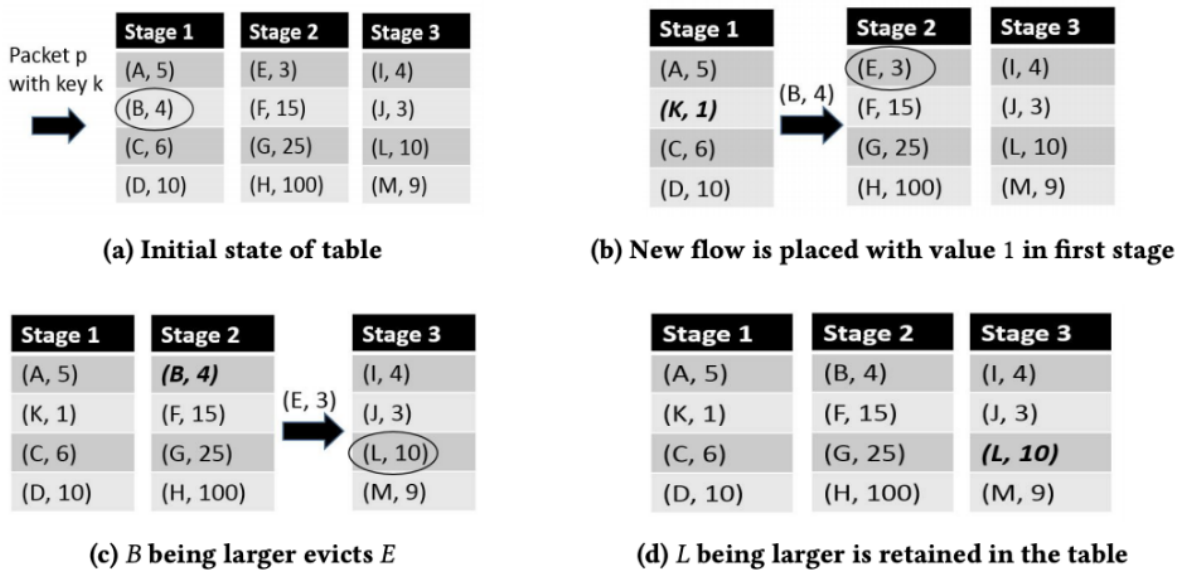

Figure 3: Illustration of hashpipe algorithm [9]

Source IP	Bloom Filter	Counter
	10110001	

Figure 4: An example of a table entry in our algorithm

4.3 Algorithm Procedure

4.3.1 General Cases. Inspired by both Hashpipe and PRECISION algorithms, we use a hash-based multi-stage table to track the superspreaders. The procedure of the algorithm can be described as follows. When a packet arrives at a programmable switch, it will first go through d-way associative memory access. We use d different hash functions to hash the source IP of the packet to an entry on the table. There are three possible outcomes for each hash. First, the hash function hashes the source IP to an entry that is empty in the table, in that case we insert to this entry by setting the source IP, update Bloom Filter and set the counter to be one. Second, the hash function hashes the source IP to an entry that is occupied, but the source IP of that entry matches the source IP of the packet. In this case, we update Bloom Filter and the counter according to whether the destination of the packet is already in the Bloom Filter. Third, the hash function hashes the source IP to an occupied entry, and the source IP on that entry does not match the source IP of the packet. Notice that the previous two outcomes are normal and requires no further attention. The third outcome however, is the hash collision case. Algorithm 1 presents the pseudo-code for how to deal with the first two general cases.

Algorithm 1 Superspreader detection algorithm in data plane for general cases (no hash collision)

```
1:  procedure ssDetection
2:      matched_i ← false
3:      for i ← 1 → d do
4:          l_i ← h_i(iSrcIP)
5:          if table[l_i] is empty and matched_i = false then
6:              Insert new entry to table[l_i]
7:              matched_i ← true
8:          if table[l_i].iSrcIP = iSrcIP then
9:              Update BloomFilter and Increment counter
10:             matched_i ← true
```

4.3.2 Recirculation to Replace Minimum Count. If all d hash functions turn out to be outcome 3, we need a way to update the table entries. The idea is to replace the entries with minimum count value out of the d entries we hashed to. We keep a minimum count $carry\_min$ and the stage number $stage\_num$ (out of d) of the minimum count in packet metadata. When there is a hash collision, we record the counter of collided entry on the table to $carry\_min$ and the stage to $stage\_num$. In the end, we will have information about which stage contains the minimum counter. However, since we have already moved on from the stage with the minimum counter, we need to recirculate a cloned version of the packet to update the table entry. This ensures that the packet will not be out of order for the next hop if recirculation is required. Similar to [2], we recirculate with a probability introduced in Equation 1.

$$p = \frac{1}{carry\_min * 100} \tag{1}$$

The *carry_min* is the minimum counter value (number of distinct destinations) to replace. If the source we want to replace contacted a lot of distinct destinations, it is more likely to be a superspreader therefore we want it to stay in the table. This probability ensures that such source would be less likely to be replaced. The actual theoretical proof of this probability is beyond the capabilities of the authors. However, the proof in [2] and our empirical result show that such probability definition provides promising performance of our algorithm. To implement such probability in P4, we generate a bit of length $carry\_min * 100$ and only recirculate if that value is zero.

There is a problem, however, with recirculation using this probability. Given that most source IP connects with only one destination in our dataset, many table entries would contain a count value of one. The probability of recirculation for such entry is $p = 1/100$. Given the amount of data, this can lead to very high number of recirculation, thus decreasing the throughput of our algorithm. Moreover, if we find a count value of one, this is already a minimum count and there is no need to go through the next stage to track the minimum count. Therefore, when hash collision occurs and the collided entry has a count of one, we directly replace that entry with the source IP of the packet. This drastically decreases the number of recirculation in our algorithm and increases the throughput of the switch. Algorithm 2 shows the pseudo-code of our algorithm with recirculation for minimum replacement added.

4.3.3 Recirculation for Duplicate Removal. Even though directly inserting when counter is one reduces the number of recirculation required, it introduces new problems of duplicate entries. Consider the following example. At $d = 1$, a source IP A is hashed to an entry whose source IP is C with a count of two. A records that entry as the current minimum. At $d = 2$, A is hashed to an empty entry and is inserted into the table. After several new packets come, the entry at $d = 1$ with source IP C is replaced by a source IP B with count of one due to recirculation to replace min. As a result, a new packet with source IP A comes in and it will directly replace the entry at $d = 1$ with source IP C. Now we have two As in different table entries. Duplication is bad because it takes the sacred memory resources and thus decrease the accuracy of our program. Even though it is a unlikely event, we implement a duplication removal method to solve this issue. It works as follows: after a packet is matched, it will go through the rest of the stages and check if there is a duplicate entry. If there is a duplicate entry, the packet will be cloned and recirculate back to remove entries with the smaller count. Algorithm 3 shows the complete pseudo-code of our algorithm with duplicate removal.

Even though we have more recirculation than the original PRECISION paper, our empirical result has shown that the recirculation rate is as low as about 20 per 300,000 packets.

4.3.4 P4 Implementation. We wrote and compiled a P4 program prototype (400 lines) of our algorithm. It is written in P416 and compiled by behavioral model. Running the algorithm on a physical programmable switch is left for future work. This shows that our algorithm can be indeed implemented in the data plane.

Algorithm 2 Superspreader detection algorithm in data plane with min replacement

```
1:  procedure ssDetection
2:      matched_i ← false
3:      for i ← 1 → d do
4:          l_i ← h_i(iSrcIP)
5:          if table[l_i] is empty and matched_i = false then
6:              Insert new entry to table[l_i]
7:              matched_i ← true
8:          if table[l_i].iSrcIP = iSrcIP then
9:              Update BloomFilter and Increment counter
10:             matched_i ← true
11:         if table[l_i].iSrcIP ≠ iSrcIP and matched_i = false
    then
12:             if table[l_i].count = 1 then
13:                 //Directly remove the entry with counter of 1
14:                 table[l_i].iSrcIP = iSrcIP
15:                 Update Bloom Filter and increment counter
16:                 break
17:             else
18:                 oval_i = table[l_i].counter
19:                 if (oval_i < carry_min) then
20:                     carry_min ← oval_i
21:                     min_stage ← i
22:     if ∧_i^d not matched_i then
23:         new_val = 2^log2(carry_min·100)
24:         Generate a random integer R ∈ [0, new_val − 1]
25:         if R = 0 then
26:             Clone and recirculate packet
27:     if Packet is recirculated then
28:         i ← min_stage
29:         l_i ← h_i(iSrcIP)
30:         table[l_i].iSrcIP = iSrcIP
31:         Update Bloom Filter and increment counter
32:         Drop the cloned packet
```

5 EVALUATION

In this section, we would like to evaluate our algorithms using trace-driven simulations. We implement a java simulation model of our algorithm and then assess it with real world packet traces from an ISP link. In this section, we first describe our experiment set up and the dataset. Then we discuss the performance of our algorithm for detecting superspreaders when varying memory usage, number of stages, recirculation delay in packet level in section 5.1, Finally we conclude by comparing it with prior related algorithms in section 5.2.

Experiment Setup We detect the Top K superpsreaders using a set of traces from CAIDA[1]. It is from a 10Gb/s ISP backbone link in Chicago, recorded in 2016 and lasts 10 minutes. We divide these packet traces into a series of sub traces and each of them lasts 10 seconds, containing approximately 10 million packets and 10k unique source IP addresses. Each of these sub traces is viewed as one trial and the experimental results in this section are averaged

Algorithm 3 Superspreader Detection Algorithm in Data Plane

```
1:  procedure ssDetection
2:      matched_i ← false
3:      for i ← 1 → d do
4:          l_i ← h_i(iSrcIP)
5:          if table[l_i] is empty and matched_i = false then
6:              Insert new entry to table[l_i]
7:              matched_i ← true
8:          if table[l_i].iSrcIP = iSrcIP then
9:              if matched_i = false then
10:                 Update BloomFilter and Increment counter
11:                 matched_i ← true
12:             else
13:                 //duplicate removal
14:                 carry_min = min(carry_min, table[l_i].count)
15:                 Update min_stage to correspond carry_min
16:                 recirculate_type = dup
17:                 Clone and recirculate packet
18:         if table[l_i].iSrcIP ≠ iSrcIP and matched_i = false
    then
19:             if table[l_i].count = 1 then
20:                 //Directly remove the entry with counter of 1
21:                 table[l_i].iSrcIP = iSrcIP
22:                 Update Bloom Filter and increment counter
23:                 break
24:             else
25:                 oval_i = table[l_i].counter
26:                 if (oval_i < carry_min) then
27:                     carry_min ← oval_i
28:                     min_stage ← i
29:     if ∧_i^d not matched_i then
30:         new_val = 2^log2(carry_min·100)
31:         Generate a random integer R ∈ [0, new_val − 1]
32:         if R = 0 then
33:             Clone and recirculate packet
34:     if Packet is recirculated and recirculate_type = min then
35:         i ← min_stage
36:         l_i ← h_i(iSrcIP)
37:         table[l_i].iSrcIP = iSrcIP
38:         Update Bloom Filter and increment counter
39:         Drop the cloned packet
40:     if Packet is recirculated and recirculate_type = dup then
41:         i ← min_stage
42:         l_i ← h_i(iSrcIP)
43:         table[l_i] = empty
44:         Drop the cloned packet
```

over these trials. We assume the counter table in the switch has a time out mechanism and zeros out at the end of each trial.

Evaluation metrics As stated before, we use the accuracy rate and false negative (% ratio of superspreaders that all not reported) to evaluate our algorithms. When comparing with other related algorithms, we use the precision and false negative rate as evaluation metrics.

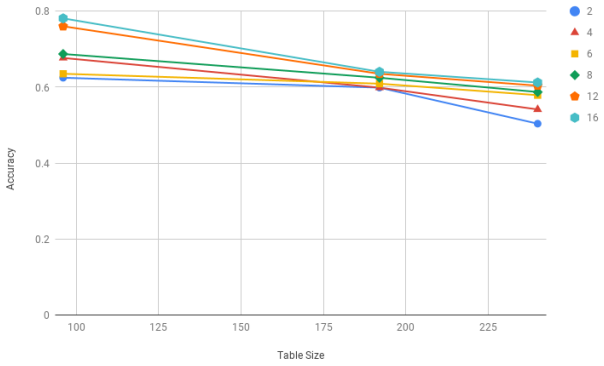

Figure 5: Effect of stage number on accuracy

5.1 Accuracy with Varying Parameters

5.1.1 Number of Stages. In this measurement, we evaluate the performance of our algorithm with various number of stages under the constraint of a fixed memory size, which means we simply divide the available memory space into more separate hash tables with the increase of stage number. There exists a trade off in the number of stages. As the stage number increases, the size of each hash table is becoming smaller, which can result in more hash collisions. On the other hand, with larger stage number, the likelihood of an incoming packet to find a smaller counter value can increase to a great extent.

Figure 5 and Figure 6 show the influence of changing stage number d on the accuracy rate and false negative. We plot the accuracy rate in the context of detecting top k superspreaders using a table with the size of k. The false negative rate is plotted for different sizes of memory m and different number of desired superspreaders s using a table with the size of k (s<k). From Figure 6, we can find that the false negative decreases with the increase of stage number for each pair of memory size and desired number of superspreaders. However, the false negative rate experiences a smooth and gradual improvement from d=4 to 16 after a sharp decrease from d=2 to 4. The results show that our algorithm manages to achieve a 5% -10% false negative rate when detecting top 20-60 superspreaders with merely 96-240 counters.

5.1.2 Reciculation Delay. As pointed before, when forcing a copy of one packet to do recirculation and go through the pipelined hash tables a second time, there can be a delay between the time when it comes into the pipeline at the first time and the time when it recirculates back and performs rewriting. During this period time, there could be other packets coming into the pipeline and changing the value of of the entry which the recirculated packet is going to modify. So the accuracy of our algorithm could be impacted by such a delay. The length of recirculation delay is decided by various factors including the length of the pipeline, the hardware architecture of the switch and the queue length of the incoming packet. Figure 7 and Figure 8 shows the impact of recirculation delay on the accuracy rate. Contrast with our concern before, We can find that our algorithm is quite robust to recirculation delay even when the delay is up to 20 packet. There could be two main reasons. First, we manage to keep the recirculation rate below 0.5%,

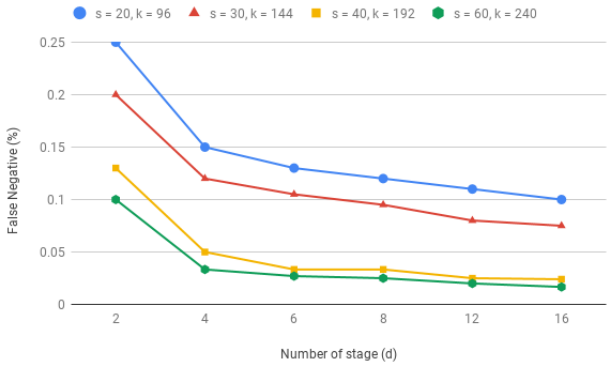

Figure 6: Effect of stage number on false negative rate

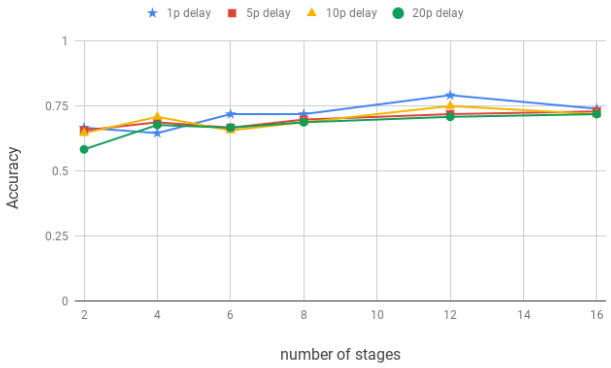

Figure 7: Effect of recirculation delay on accuracy

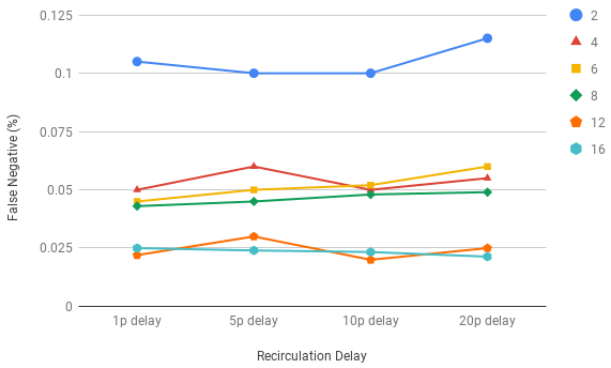

Figure 8: Effect of recirculation delay on false negative rate

which means only a very small subset of packets need to recirculate. The low recirculate rate can minimize the impact of recirculation delay. The second reason could be that we manage to eliminate all duplicate entries in the aggregated hash table, which also lowers the influences of recirculation delay.

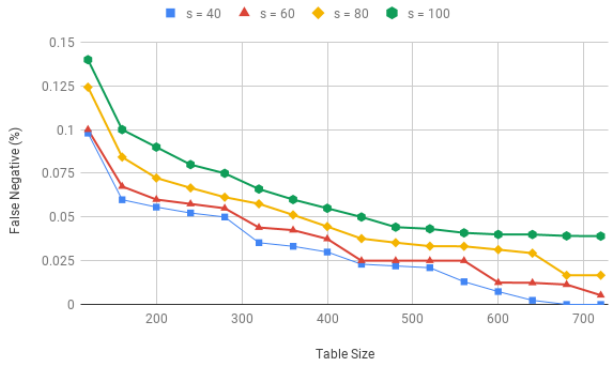

Figure 9: Effect of memory size on false negative rate

5.1.3 Memory Size. In this section, we would like to exploit the performance of our algorithm against varying memory size. Figure 9 shows the performance with the increase of memory space allocated for our multi-staged hash tables. Since the storage space for every entry in the hash table is fixed, the total memory space occupied has a linear relationship with the table size. From Figure 9, we can find that the error decreases with the increase of allocated memory size across a range of desired amount of superspreaders. Utilizing a table with the size of more than 440 entries, we manage to achieve a false negative rate below 5% with the purpose of tracking up to 100 superspreaders. It is worth noticing that for any single trial with the 10 Gib/s ISP backbone link, there are more than 2 orders of magnitude unique source IPs than the number of counters in our table.

5.2 Comparison with Existing Superspreader Algorithms

In this section, we would like to make a comparison with other superspreader algorithms including data plane algorithms and control plane algorithms. First, We would like to compare our algorithm will the ideal space saving algorithm, which can be implemented in the control plane or application level. Then we would like compare with the state-of-art algorithm for detecting superspreaders on the data plane.

5.2.1 Comparing with Space Saving. As stated before, space saving algorithm is the ideal version of HashPipe and PRECISION, which keeps a table with k entries to track top k objects. The reason why it can not be directly performed in the data plane is that it needs to sort the whole table and find the entry with the smallest counter value and then substitute. However, in a P4 switch, it is impossible to read a variety of locations at the same time. We would like to see whether the ideal space saving algorithm can achieve a good results when detecting superspreaders here. Considering the distinct destinations' characteristic of superspreader, we maintain a quite large hash set for maintaining different destination IPs and associate this hash set with a counter in a singe entry of the hash table.

We compare these 2 algorithms using same number of counters with the task of detecting Top 60 and Top 100 superspreaders.

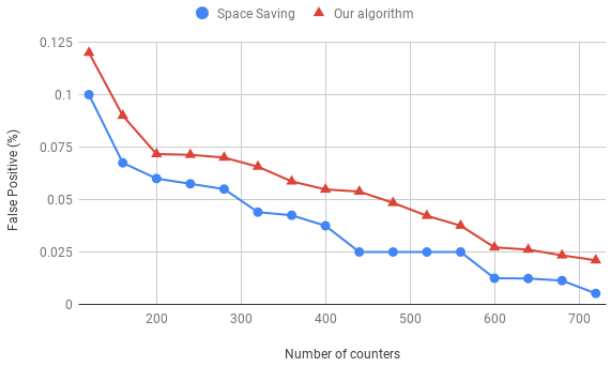

Figure 10: Performance comparison between our algorithm and space saving on Top 60 detection

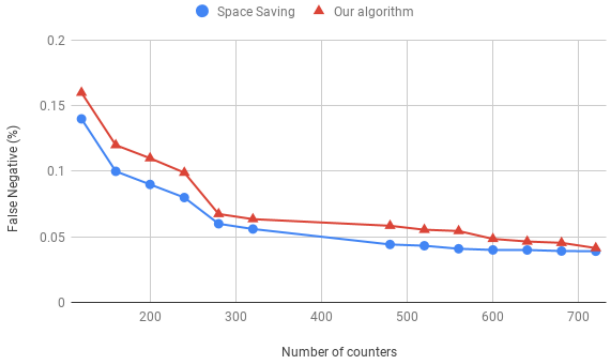

Figure 11: Performance comparison between our algorithm and space saving on Top 100 detection

As shown in Figure 10 and Figure 11, the space saving algorithm outperforms our algorithm with an decrease in false negative of 2.5% - 3%. The reason for its performance can be the following several reasons. First, the space saving algorithm does not need to do an approximate estimation of the number of destinations for each source IP since it utilize a hash set. Second, it does not to do an approximate estimation of the minimum value of counter in the whole table. However, from the figures we can also find that there is no significant performance improvement of the space saving algorithm compared with our algorithms. Moreover, considering the memory storage the space saving algorithm uses for storing distinct destination IPs is not scalable and much larger than that of our algorithm, such kind of performance decrease of our algorithm is totally acceptable.

5.2.2 Comparing with Seek and Push. As mentioned in our related work, seek and push is by far the only implemented algorithm in the data plane. It implements an Elastic Trie architecture and performs superspreader detection. However, different from our definition of top-k, they define top-k superspreaders as sources which have had connections with more than k destinations. In order to perform a

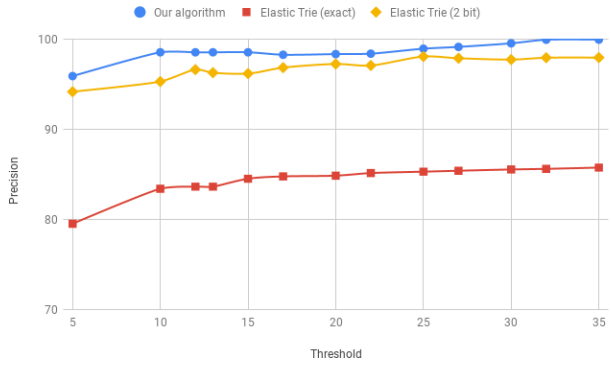

Figure 12: Precision with the change of threshold

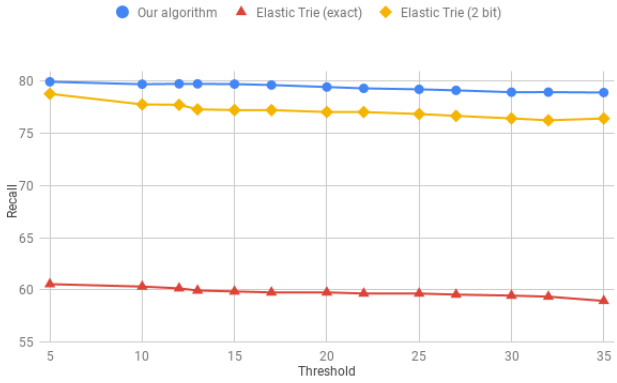

Figure 13: Recall with the change of threshold

fair comparison and align our results with one produced by their system, we aggregate our outputs and consider only those with a counter value larger than the fixed threshold k when detecting Top k superspreaders. It is worth mentioning here that the false positive rate is zero because there is no overestimation with our algorithm. Moreover, because this kind of change, we here use the recall and precision as our evaluation criteria.

Figure12 and 13 above plots the precision and recall of these two algorithms across a range of desired amount of superspreaders. It is clearly shown that our algorithm outperforms two different kinds of their algorithms both in these two criteria with a more economic usage of memory space.

6 CONCLUSION

Superspreaders are responsible for attacks such as worm propagation and DDoS. Detecting potential superspreaders can prevent such attacks. Existing superspreader detection uses data plane to collect data and control plane to run analysis. Although such method can achieve promising result, the communication overhead between the planes limit the reaction time when facing those attacks. As a result, with the emergence of programmable switches, it is feasible to develop superspreader detection algorithm entirely in the data plane. This allows the switches to react immediately when those

attacks happen (e.g. drop packets with source IP that matches an entry in the table of potential superspreaders).

In this paper, we presented a superspreader detection algorithm entirely in the data plane. To the best of our knowledge, this is the very first algorithm that solely detects superspreaders in the data plane. We take inspirations from existing data plane heavy-hitters detection and build a hash-based multi-stage table to track the potential superspreaders. In order to do this, we use a Bloom filter to keep tracks of number of distinct destinations contacted by a source IP. We limited the size of the table to prevent it from growing unbounded. Our algorithm combines and develop the ideas from Hashpipe and PRECISION and makes them specific for superspreader detection.

From the evaluation result, we find that our algorithm manages to achieve a 5% -10% false negative rate when detecting top 20-60 superspreaders with merely 96-240 counters. The performance increases as the amount of available memory increases. Moreover, we uses a probability scheme that keeps our recirculation rate low. As a result our algorithm is robust to recirculation delay.

We also compared our algorithm with existing work. In particular we compared the performance in superspreader detection with Elastic Trie [6]. We find that our algorithm outperforms the Elastic Trie in both precision and recall with our dataset. Moreover, we find no significant decrease in performance with the idea Space Saving algorithm.

We wrote and compiled a P4 version of our algorithm, showing that it can be indeed implemented in the data plane. An interesting future work would be to run our P4 program in a physical programmable switch to validate our implementation. Another interesting future work would be to compare our algorithm with those running on the control plane such as OpenSketch. We believe our algorithm can perform at the same level of accuracy while having the benefit of reducing communication overhead.

REFERENCES

[1] 2016. The CAIDA UCSD Anonymized Internet Traces - 2016, http://www.caida.org/data/passive/passive_dataset.xml.

[2] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, Cambridge, UK.

[3] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, Vol. 13. ACM, New York, NY, 422–426.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walke. 2014. P4: Programming Protocol-Independent Packet Processors. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, New York, NY, 87–96.

[5] Kreutz D, Ramos F, and Verissimo P. 2013. Towards secure and dependable software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM, Hong Kong, China, 55–60.

[6] Jan Kucera, Diana Andreea Popescu, Gianni Antichi, Jan Korenek, and Andrew W Moore. 2018. Seek and Push: Detecting Large Traffic Aggregates in the Dataplane. In *eprint arXiv:1805.05993*.

[7] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM, Florianopolis, Brazil, 101–114.

[8] Patrick P. C. Lee Runhui Li Lu Tang Yi-Chao Chen Gong Zhang Qun Huang, Xin Jin. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM, Los Angeles, CA, 113–126.

[9] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research*. ACM, New York, NY, 164–176.

[10] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software defined traffic measurement with OpenSketch. In *nsdi'13 Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. USENIX, Lombard, IL, 29–42.