Flows Passing in the Night: A Reproduction of "Heavy-Hitter Detection Entirely in the Data Plane"

Sierra Kaplan-Nelson
Stanford University
Stanford, CA
sierrakn@stanford.edu

Colin Kincaid
Stanford University
Stanford, CA
ckincaid@stanford.edu

ABSTRACT

Sivaraman et al.'s paper "Heavy-Hitter Detection Entirely in the Data Plane" proposes HashPipe, a heavy-hitter detection algorithm intended for implementation in programmable switches. HashPipe tracks heavy-hitting TCP flows in a multi-stage hash table, evicting candidates when heavier flows are observed. The paper finds that it is possible to decrease HashPipe's false negative rate by increasing both the hash table's memory size and the number of stages in the table, with only minor increases to the amount of duplicate flows stored in the table as a side effect. We attempt to reproduce the paper's key figures, using our own implementations with the original network traces. Our findings corroborate some, but not all, of the original paper's results. We explain why our results might make more sense than the original paper's well-behaved trends, and we offer a proposal for obtaining better results when the network being studied has a relatively low ratio of distinct hosts to total flows.

1 INTRODUCTION

At the granularity of a single switch, it is useful to detect TCP flows that contribute a disproportionate amount of network traffic. Identification of these "heavy hitter" flows can be used to prevent denial-of-service attacks, reduce congestion through traffic engineering, cache frequently used forwarding table entries, and more. Heavy hitter flows are defined for our purposes as the top k flows ranked by number of constituent packets, where application designers choose the appropriate value of k for their purposes.

Some algorithms to detect these heavy flows involve sampling packets at the switch and sending them to a server for analysis, as in NetFlow[1] and sFlow[5]. Since packet sampling is expensive, these solutions typically only inspect 0.01% - 0.1% of a switch's received packets. Operating on such a small flow sample can lead to inaccurate heavy hitter classification, and the requisite server communications contribute undesirable network traffic. Other algorithms classify heavy hitters by storing each received packet in a data structure. Existing data structures for this problem are either too memory-intensive (such as a counter, whose storage scales with the total number of flows) or too computation-intensive (such as sketches, which require significant overhead in order to support flow identifier extraction[2]) to be implemented in a switch. Even the space-saving algorithm—designed to address both of these issues—requires searching a table[3], an operation too expensive for a switch's packet processing path. None of these options is feasible for an algorithm meant to run in the data plane.

Sivaraman et al. describe HashPipe, the first algorithm to use programmable switches to identify heavy hitters while operating at line rate, offering improved accuracy over sampling approaches and memory overhead proportional to the specified number of heavy hitter flows[4]. HashPipe caters to switch architecture processing constraints by splitting the space-saving algorithm's hash table into multiple stages, where each stage requires only a small number of constant-time operations. This gives HashPipe the unique ability to operate entirely in the data plane.

2 METHODS, DESIGN, AND EXPERIMENTAL SETUP

We first use TShark to convert traces of real network traffic in the form of pcap files into a Python-friendly CSV format. From each TCP packet in the trace, we extract header fields that identify the flow to which it belongs. We use these identifiers to simulate running packets through a P4 switch, where the HashPipe algorithm operates.

In each phase of the algorithm, a packet's flow identifier is hashed to a slot in a phase-specific hash table (alternatively described as one "stage" of a multi-dimensional hash table). The flow will create a new entry in that slot if it is currently empty or merge with the existing entry if it corresponds to the same flow; otherwise, the new flow will evict the current entry if the new one is more of a heavy hitter, but in either case, the flow not in the table after the comparison will be carried to the next stage. Each stage of the table is independent from the others, so once one packet has been fully processed at the first stage, the next packet can begin processing—allowing the whole algorithm to operate at line rate. The k heaviest-hitting flows are identified across the whole table after all packets have been processed.

The original paper identifies each flow by its source IP address, so we do as well. Sivaraman et al. note that they could identify flows more accurately by using more of its header fields: some larger subset of its source IP address, destination IP address, source TCP port, destination TCP port, and protocol number. A more granular identification of flows means that the algorithm deals with a larger number of distinct flows—which is what we want—but that also means that we need more table slots (and thus more memory) to achieve comparable accuracy. We discuss this tradeoff further in Section 3.4.

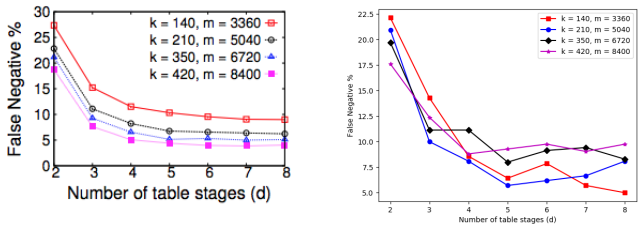
For ease of creating a test harness, we build a HashPipe simulator in Python. After verifying that the simulator produces the exact same tables as our line-rate P4 program on several traces, we use the simulator to assess the algorithm's accuracy.

We quantify the fidelity of our heavy hitter detection by measuring the percentages of false positives (flows erroneously identified as heavy hitters), false negatives (heavy hitter flows that are not detected), and duplicate flow entries across the table, trying to match the original paper's results with various values of configuration parameters d, k, m (table stages, number of heavy hitters, and memory allocation). We also seek to replicate the original paper's finding that the heavy hitter flows not identified by the algorithm (false negatives) are less heavy than those correctly identified. To verify this result, we plot the percent of false negatives against a varying value of k, the number of flows we classify as heavy hitters. The exact graphs we reproduce are Figures 2, 3, and 6 from the original paper, using the same ISP backbone link trace and datacenter trace.

As a note, we wrote all of the code for these experiments ourselves. That said, we had access to the authors' repository and did refer to their Java simulator and P4 code throughout the process. Their co-prime coefficients for generating d-independent families of hash functions for the different table stages were particularly helpful.

3 RESULTS AND EVALUATION

In the following sections, we show the results of our reproduction attempts. We omit our false positive experiments as the percentages are so small as to not be interesting. The total number of flows is over 400,000 in the ISP trace and about 5,000 for the datacenter trace. There are roughly the same number of packets for each trace. Although we run our code on the data from the same traces as the original paper, the exact sections of the traces that we chose may have been different, explaining some of the disparities in our results.



(a) Original Figure 2a **(b) Our reproduction**

Figure 1: Comparison of ISP data false negatives

3.1 False negatives with ISP data

In Figure 1, we see that our false negative percentages for the ISP experiment match those of Figure 2a from the original paper in general trend. However, there are a few noteworthy differences:

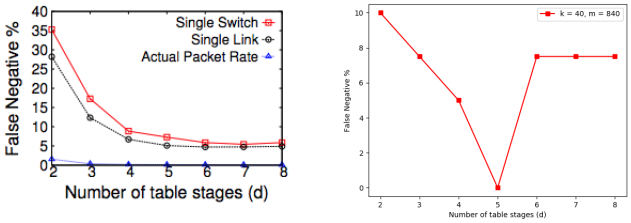
- For each value of k and m, our percentage of false negatives does not decrease monotonically with increasing number of table stages.
- The false negative rate for each line is lower at the second stage in our reproduction than in the original figure.
- At each stage past the first, configurations with more memory sometimes yield higher false negative rates than configurations with less memory.

We do not compare our implementations to the original code in order to figure out why our results differ, but it makes sense to us that the percent of false negatives would not necessarily decrease after a certain number of stages. The addition of third and fourth stages are important for evicted flows to find their way back into the table, but past that, all that really changes when we add more stages are the specific pairwise flow comparisons made at each table stage. One can imagine that even with eight stages, a legitimate heavy hitter flow might get evicted because of hash collisions with heavier hitters even though it would be correctly identified with a smaller number of table stages (when different collisions occur). As we show in section 3.5, the false negatives that contribute to the percentages in Figure 1 are indeed the least heavy of the heavy hitters, meaning they could be evicted as a result of unlucky hash collisions.

Our reproduction does corroborate the original paper's important finding that the percentage of false negatives decreases significantly upon increasing the number of table stages beyond two or three. It seems strange to us that according to our results, the simulation with the least memory does the best among all simulations when we reach a high number of stages. We suspect this is due to the fact that the simulation also has a lower number for k, so if the top 140

heavy hitters are much heavier than the next 140, we would expect to see the false negative rate increase with k because borderline heavy hitters are more likely to be erroneously evicted.

3.2 False negatives with datacenter data

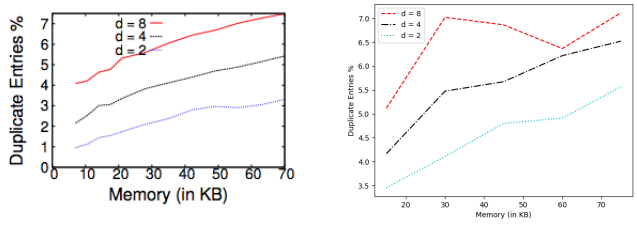


(a) Original Figure 2b **(b) Our reproduction**

Figure 2: Comparison of datacenter data false negatives

Figure 2 shows our attempt at replicating the HashPipe simulation on the datacenter trace. Our results are difficult to compare to the original paper's because we only ran the "actual packet rate" experiment and did not isolate the data for a single link or switch. With that in mind, the percent of false negatives decreases with the number of stages until reaching 0% at five stages, which matches the paper's results. After five stages, we begin to see false negatives again—which the original experiment does not. We rationalize this result with the same pairwise comparison argument as we make for the ISP data results.

3.3 Comparison of duplicates with ISP data

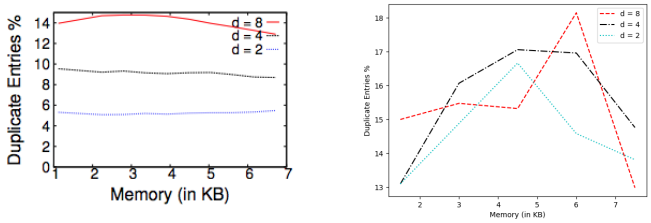


(a) Original Figure 3a **(b) Our reproduction**

Figure 3: Comparison of ISP data duplicates

Figure 3 shows the highest fidelity reproduction we achieve among our experiments. The percent of duplicate entries in the table (mostly) increases with the amount of memory. It makes sense that having more stages in the table would increase the percent of duplicate entries, because having more stages increases the number of opportunities for a duplicate flow identifier to settle in an early stage of the table instead of merging with an existing entry at a later stage. The percent of duplicate entries in both tables is never more than 7%, but in our table the lowest percent of duplicate entries is 3%, which differs slightly from the original experiment's minimum of 1%. One minor detail to note is that the original figure's lines look more continuous than ours because they ran the experiment with more distinct memory values.

3.4 Comparison of duplicates with datacenter data



(a) Original Figure 3b **(b) Our reproduction**

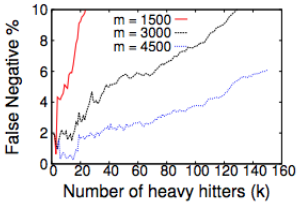
Figure 4: Comparison of datacenter data duplicates

Our reproduction of the duplicate-measuring experiment with the datacenter data, seen in Figure 4, shows more erratic results. Our percentages of duplicate entries are higher, though the ranges are similar. More stages and more memory translate to more duplicate entries in general, although there is a strange dropoff after 4-6KB. We were not able to come up with a good explanation for this phenomenon, but we make a separate observation that basically invalidates the experiment in the first place: Our attempt to measure false negative or duplicate rates for datacenter data, where there are many distinct flows but relatively few distinct IP addresses because the network is well-connected, reveals the need for flow identifiers more descriptive than just the source IP address. We achieve low false negative rates, but at the cost of not identifying heavy hitters in a meaningful way. We expect that datacenters will mostly want to use heavy hitter information for traffic engineering and forwarding table entry caching, neither of which can be done when flows are only identified by the source IP address. Experiments using the datacenter data should not be taken too seriously until this major shortcoming is addressed.

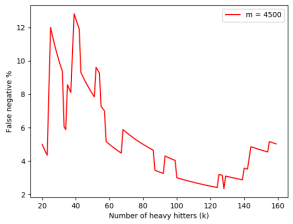
3.5 ISP data comparison of false negative rates with a varying number of flows classified as heavy hitters

The original paper observes that the heavy hitter flows that are not correctly identified tend to be the less heavy ones. They demonstrate this with a graph showing that the percent of false negatives increases with the value of k (where m and d are held constant). Concretely, if the number of heavy hitters we try to identify increases from k to k' and the percent of false negatives increases, that means that a flow among the least heavy (k' - k) flows are more likely to be misidentified than a flow in the k heaviest flows.

When we first tried to verify this result using the ISP trace and the same values of m, k, and d as the original paper, we were unable to reproduce the same results (Figure 5). However, in Figure 6, we observe that by increasing k, we can make the percent of false negatives follow the same gradually increasing trend observed in the original paper. We attribute the need to increase k to whatever factor(s) caused the differences observed in the other experiments and are satisfied seeing that the trend ultimately holds up.

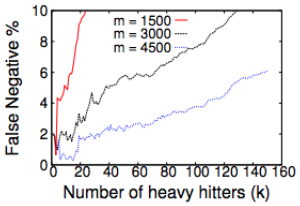


(a) Original Figure 6a

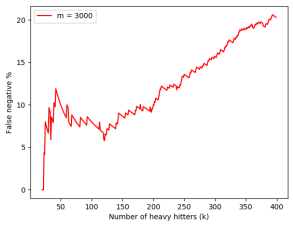


(b) Our reproduction (attempt 1)

Figure 5: Observation that less heavy hitters are more likely to be false negatives in ISP data (unclear trend)



(a) Original Figure 6a

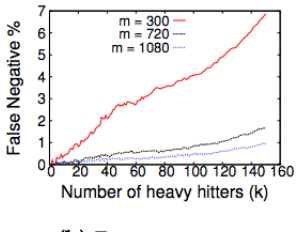


(b) Our reproduction (attempt 2)

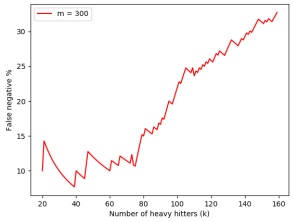
Figure 6: Observation that less heavy hitters are more likely to be false negatives in ISP data (made clear by inflating the number of heavy hitters)

3.6 Datacenter data comparison of false negative rates with a varying number of flows classified as heavy hitters

Our attempt to reproduce the same finding with the datacenter data was likewise successful. Our results follow a similar trend, with an early peak in the percentage of false negatives and then a steady increase as k increases. Overall, we believe these results verify the original paper's claim that less heavy flows are more likely to be misidentified as not being heavy hitters than heavier flows.



(a) Original Figure 6b



(b) Our reproduction

Figure 7: Observation that less heavy flows are more likely to be false negatives in datacenter data

4 CONCLUSION

In this paper, we confirmed that the HashPipe algorithm does indeed classify a large percentage of flows correctly as heavy hitters, even when there are over 400,000 flows and 100 times fewer memory slots in the hash table. Most importantly, we showed that this algorithm can be tweaked to produce better results for different scenarios by changing the number of stages in the hash table, the overall amount of memory, and the number of heavy hitters desired. We showed that for an ISP backbone trace, the IP source address is generally good enough to identify a flow; however, for a trace from a well-connected network such as a datacenter, more granular identifiers are needed to classify each flow in order to produce meaningful results. All of this can be achieved in a switch's data plane at line rate using the P4 language.

The discrepancies between our results and the original paper's likely arise from differences in pairwise flow comparisons, which are made when two flows have a hash collision at some stage in the table. This leads us to suspect that the major differentiator between our simulator and the original authors' lies in our hash functions. It is entirely possible that we misunderstood how to implement the hash functions, but for the reasons outlined at the end of Section 3.1, we actually believe that the original paper's well-behaved results

(with fewer local maxima and minima) make less sense than ours. Future attempts at reproduction would benefit from careful reasoning about the algorithm's hash functions and their mutual independence, which may become even more important if more granular flow identification is done.

5 ACKNOWLEDGMENTS

We would like to thank Vibhaa Sivaraman, author of the original paper, for her prompt and thorough advice. Vibhaa's guidance saved us from going down several unnecessary rabbit holes throughout this reproduction. We would also like to thank Jennifer Rexford for putting us in touch with Vibhaa.

REFERENCES

[1] IOS Cisco. 2008. NetFlow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netow/index.html.. (2008).

[2] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[3] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*. Springer, 398–412.

[4] 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, 164–176.

[5] Mea Wang, Baochun Li, and Zongpeng Li. 2004. sFlow: Towards resource-efficient and agile service federation in service overlay networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*. IEEE, 628–635.