Herding the Elephants: Detecting Network-Wide Heavy Hitters with Limited Resources

Anonymous Author(s)

ABSTRACT

Detecting "heavy hitters" e.g., flows with traffic volume exceeding a threshold, is the core of many network monitoring applications. While past work has shown how to measure heavy hitters on a single switch, network operators often need to identify *network-wide* heavy hitters on a small timescale to react quickly to anomalies. Detecting network-wide heavy hitters efficiently requires striking a careful balance between the memory and processing resources required on each switch and the network-wide coordination protocol.

We present Herd, a distributed technique for detecting network-wide heavy hitters with high accuracy under communication and state constraints. Our solution combines the sample-and-hold algorithm for measurements on the switches with probabilistic reporting to a central coordinator. Based on these reports, the coordinator adapts the reporting threshold and probability at each switch to the spatial locality of the flows. We present an algorithm to tune Herd in order to maximize detection accuracy under resource constraints. Simulations using real traffic traces show that our P4-based prototype can detect network-wide heavy hitters accurately with 17% savings in communication overhead and 38% savings in switch state compared to existing approaches.

1 INTRODUCTION

To effectively manage their networks, operators continuously monitor their traffic to detect attacks, congestion, and failures. To identify these conditions, operators often seek to detect *heavy-hitters* by separating the *elephant* flows from the *mouse* flows. Elephants are the relatively few flows that significantly contribute to the overall traffic volume and mouse flows are the more numerous but smaller flows. To distinguish flows into these two categories, network operators must choose between measuring flows in network devices, which have both limited memory and computational capacity, or sending a subset of traffic to general-purpose CPUs for analysis. In the latter case, the operator is limited both in how much traffic can be sent across the network but also by the fact that general-purpose CPUs process data far slower than line rate.

Currently deployed solutions, such as NetFlow [6] and sFlow [18], rely heavily on packet sampling (e.g., sampling 1 out of 4000 packets [14]) to reduce the measurement data exported from the switches. Unfortunately, low sampling rates result in longer delays for computing accurate estimates of flow sizes, which lead to temporal "blind spots" to short-term network conditions (e.g., TCP incast [5] or microbursts). Other solutions [1, 15, 23, 28] run streaming algorithms (with compact data structures like count-min sketch [8]) directly on the resource-constrained switches. These techniques detect heavy hitters on each switch individually. Flows that generate a large volume of traffic for the network in total, but are not heavy at any single switch would go undetected.[1]

For example, if a host inside the network is the victim of a denial-of-service attack and monitoring is performed at the ingress switches, each ingress switch may only observe a moderate amount of traffic to the victim host, yet aggregating the analysis over all switches would indicate that a high volume attack is occurring. Furthermore, when attacks of this magnitude begin to converge at a network choke point or at the victim itself, network devices can become unresponsive which may prevent further measurement from being performed and stymie root-cause analysis. Therefore, it is appropriate to place the monitoring upstream of convergence, i.e., at multiple locations which are able to handle fractions of the overall attack.

Detecting network-wide heavy-hitters reduces to a distributed monitoring problem among edge switches, i.e., the entry points of traffic into the network, and a centralized coordinator. The coordinator aggregates the partial information observed at each switch to identify flows whose aggregate count exceeds a global threshold. Deciding when a switch should report to the coordinator and what the switch should report determines how much communication is required and, ultimately, the accuracy of the results. For example, recent work [2, 14] shows how to periodically collect and combine sketches from multiple locations to compute a network-wide estimate of the traffic. However, these techniques fail to strike the balance between *real-time* analysis and low communication overhead.

The Continuous Distributed Monitoring (CDM) model provides a communication-efficient method for reporting local conditions, as-needed, to continuously track the heavy hitters without respect to a fixed monitoring interval. However,

CoNEXT '19, December 9-12, 2019, Orlando, Florida, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Here, the definition of a flow may be more coarse-grained than a five-tuple, such as source/destination IP address.

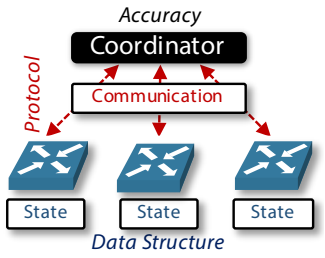


Figure 1: Herd Architecture. The coordinator aggregates the partial information observed at each switch to identify network-wide elephants.

this model assumes the ability to store per-flow state, which has hindered its adoption in actual monitoring systems [7].

We present Herd (depicted in Figure 1), a practical monitoring system for detecting network-wide heavy hitters in real time, with high accuracy, and under communication and state constraints. We extend the standard taxonomy of mice and elephant flows to more accurately describe the costs of performing heavy-hitter detection in a network-wide setting. Herd instructs each switch to probabilistically identify and report potentially substantial flows, while accounting for the locality of flows to minimize communication. Our solution extends probabilistic reporting techniques presented in [27] and combines them with the sample-and-hold algorithm [10], to report measurements of individual flows in real time. Herd tunes the system parameters using representative traffic observed by the network to achieve the best accuracy possible within the available memory and bandwidth constraints. We summarize our contributions as follows:

Continuous, communication-efficient coordination. We developed a new coordination protocol for detecting network-wide heavy hitters that uses adaptive thresholds to account for flow locality. This protocol probabilistically reports when switches observe a non-trivial contribution from a monitored flow and infers network-wide heavy hitters at the coordinator from these reports. Our analysis shows that this protocol reduces the communication cost by 17% for achieving 97% accuracy compared to sampling.

Memory-efficient switch data structure. We developed a data structure that efficiently stores locality parameters and counters for flows that show a non-trivial contribution to a network-wide count. This data structure probabilistically determines the subset of flows to monitor at the switch from a larger traffic stream. We demonstrate that this data structure can be implemented in modern programmable switches for line-rate execution. Our evaluation shows that our solution requires 40% less switch memory at the expense of 3% degradation in detection accuracy when compared to counting all of the flows.

Parameter-tuning algorithm for high accuracy. While our solution consists of well-known algorithmic [9] and data-structure [10] building blocks, combining these building blocks to produce accurate results within resource constraints is challenging. We present an algorithm that relates the parameters of both the protocol and the data structure to the potential significance of flows to be monitored and how those parameters affect Herd's performance in terms of accuracy, communication, and state. We describe a heuristic for achieving high accuracy under communication and state constraints.

In § 2, we summarize Herd's architecture for network-wide heavy hitter detection. We present the design of the coordination protocol in § 3, and the switch data structure in § 4. In § 5, we present an algorithm for configuring various system parameters. To demonstrate the deployability of Herd, we present our Protocol Independent Switch Architecture (PISA) [3] prototype in § 6. Finally, we evaluate our protoype in § 7, and discuss related work in § 8.

2 HERD OVERVIEW

The Herd architecture is composed of ingress switches and a centralized coordinator, as shown in Figure 1. The coordinator aggregates the partial information observed at each switch to identify flows whose aggregate count exceeds a threshold. By counting locally at each switch and periodically reporting to the coordinator, we can reduce the communication cost, but the memory at switches is limited which also affects accuracy. In this section, we first describe the taxonomy of flows in the network that affects how much state and communication is required to perform network-wide heavy-hitter detection and then we describe the mechanisms we use to distinguish flows in that taxonomy.

2.1 Who's Who in the Zoo

In order to identify network-wide heavy hitters under communication and state constraints, we need a way to classify flows both *locally* and *globally*. Additionally, we need to be able to relate the sizes of these classes to the amount of memory and communication required to perform the network-wide detection.

Local moles and mules. Global elephants. We extend the traditional taxonomy of flows by introducing two new classes: *moles* and *mules* (see Figure 2). At each switch, a large number of flows will have no local or global significance which is the traditional class of mice; we seek to allocate no scarce resources for these flows. However, a smaller set of flows may be significant locally, but a switch will not know if these flows matter globally. Switches will have to maintain state for these flows to determine whether or not they *might* have global significance; we call these flows moles. However, when a mole reaches a local threshold that could significantly

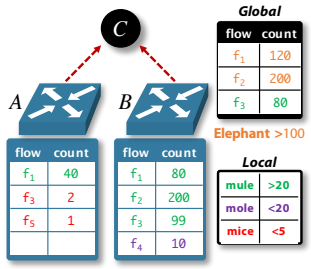


Figure 2: Example zoo thresholds and counts.

impact a global count, a switch is obligated to inform the central coordinator; we call these flows mules.

Mules are, inherently, tracked both locally and globally. A single switch that determines a flow is a mule and reports it to the central coordinator, which then tracks the flow as a mule globally. Based on the reports sent by the switches for each of their mule flows, the central coordinator determines when a mule flow has become a network-wide elephant.

The zoo determines resource allocation. This extended taxonomy allows us to better reason about the resources required to perform network-wide heavy-hitter detection. With a known upper bound on the state per switch, we focus our effort on ensuring that the number of moles at each switch does not exceed the upper bound. Similarly, if we know the allowed communication rate for each switch, we must ensure that the number of mules and their reporting frequency does not send too many reports to the coordinator. In the next section, we discuss the mechanisms Herd uses to distinguish among these classes of flows.

Example. Figure 2 shows a simple example where two switches (A and B) communicate with a central coordinator (C) to determine the network-wide elephants. In this example, we use thresholds of 5 and 20 to distinguish mole from mouse flows and mule from mole flows, respectively; we set the threshold for network-wide elephants at 100. The tables below each switch show the actual counts observed for flows f_1-f_5 at each switch. At switch A, we avoid maintaining state for mouse flows f_3 and f_5, but store the counts for the local mule flow f_1. At switch B, we store counters for f_1-f_4, but we do not report f_4 to the coordinator because it is only a local mole. The coordinator is aware of all mules (f_1-f_3) from both switches, but determines that only f_1 and f_2 are global, network-wide elephants. In the case of f_3, notice how both switches and the coordinator all have *different* views of the total count for this flow. Since f_3 is a mouse at switch A, the switch actually has no information about the flow's count at all. At switch B, flow f_3 is a mule locally, but since the switch reports to the coordinator only once every 20 counts (the mule threshold), the coordinator believes the global count of f_3 is only 80. In fact, the global count of f_3 meets the network-wide threshold of 100, but our taxonomy of flows and their reporting requirements would not identify f_3 as a network-wide elephant; this is by design.

2.2 Probabilistic Counting and Reporting

Based on the above taxonomy, we must distinguish mice from moles and moles from mules. Once differentiated, we must also determine how frequently to update the coordinator with local information about the mules. In this section, we describe the techniques Herd uses for doing so.

Distinguishing moles from mice. Many existing techniques for storing flow counters with small state focus on accurately detecting only the local heaviest flows. For example, using a count-min sketch does not *eliminate* storing state for small flows, but it does provide bounds on the error incurred by doing so. Using a count-min sketch would both violate our goal of maintaining no state for mice, but we would also need a very large sketch to overcome the error incurred by storing the small but numerous mouse flows. Similarly, the space-saving algorithm [16] works well for storing local elephants, but that algorithm would allow local mice to evict moles from the data structure, which would lead to inaccurate results.

To avoid maintaining state for small flows, we use a data structure that relies on the sample-and-hold technique [10]. In this technique, the switch checks whether each incoming packet belongs to the set of moles. If so, it updates the counter; otherwise, the switch chooses to start counting the flow with some sampling probability (s). Effectively, this approach defines the set of moles as those whose count is greater than $(1/s)$, in expectation. We show how to choose s such that it reduces the memory footprint without compromising the detection accuracy in Section 4.1.

Distinguishing mules from moles. Only a subset of the mole flows will ever become large enough to impact the global count. We set a local threshold (τ) for local flow counts such that $1/s < \tau$. This ensures that the set of mule flows is strictly smaller than the set of mole flows. When a local mole flow's count reaches τ, the switch promotes the flow to a mule locally and reports to the coordinator.

Reporting Mules. Requiring all switches to send reports each time (τ) packets are observed for all mule flows to the coordinator would limit our system's ability to support large networks with many ingress switches. Rather than sending a report for each mule every time τ packets are observed, we build on the theoretical approach first described in [27] and report to the coordinator with probability (r) each time (τ) packets are observed for a mule. The coordinator identifies a mule as a network-wide heavy hitter if it receives R reports for this flow from any of the switches. We could choose to report very frequently (e.g., $r = 1$) for high accuracy, or we could choose a lower value of r to lower the coordination

overhead. In Section 3.1, we show how to select values for τ, r, and R that strike a balance between detection accuracy and communication cost.

3 COORDINATION PROTOCOL

The coordination protocol must allow the edge switches to efficiently communicate to the coordinator when they have observed counts that could be significant network-wide. In this section, we describe a protocol that sets the threshold for each mole flow to become a mule, and then uses the reports about the mules to determine the network-wide elephant flows. We then describe an extension to this protocol that leverages the spatial-locality of network traffic to reduce the communication cost of the protocol.

3.1 When to Report Which Flows

3.1.1 Separating the Mules from Moles. The switch distinguishes locally between moles and mule flows, and in doing so it determines *which* flows to report. The switch can perform this discrimination by comparing the count of a mole to a local threshold (τ) set by the coordinator. Once a mule flow is identified, the switch reports to the coordinator, each time a *bundle* of τ packets is observed at the switch—answering *when* to report.

3.1.2 Scaling to Large Networks. However, the downside of the above technique is that it will not scale as the number of switches in the network grows. Because a switch only reports a flow once for every τ packets it observes, it will often have residual flow counts smaller than τ which have not yet been reported. In aggregate, these residual counts represent a "blind spot" for the coordinator and necessarily cause inaccuracies in the global count it maintains.

Algorithm 1: Switch Algorithm

```
Input: Local Threshold (τ), Report Probability (r)
Func ProcessPacket(pkt):
    f ← ExtractFlow(pkt)
    exceeds ← UpdateAndCheck(f,D)
    if exceeds
        if Flip(r)
            Report(f)
        D[f] ← 0
```

As τ increases or the number of switches grows, the inaccuracy of the final results will increase. One possible way to reduce the inaccuracy is to significantly lower τ. However, that would increase communication, since the switch will produce many more reports for each mule flow. Prior work [9] proposed a probabilistic reporting approach that scales with the number of switches in the network and proved its efficiency. However, implementing this technique has proven to be challenging, and has yet to be implemented [7]. We adapt this technique to account for flow locality and enable execution on modern programmable switches.

3.1.3 Probabilistically Separating Elephants and Mules. Our algorithm for probabilistically reporting mule flows to the coordinator is described in Algorithm 1. The function `ProcessPacket` processes every packet received by the switch and `ExtractFlow` extracts from the packet the fields that identify flow f. The function `UpdateAndCheck` updates the counter for this flow and compares it with a local threshold (τ) (see Section 4 for further details). If the updated count exceeds τ, then the switch reports the flow to the coordinator with probability r. Here D is just a simple key-value store and a single r and τ apply to all flows. By reporting with probability r, each bundle reported now represents a count of τ/r in expectation, which reduces the total number of reports that must be sent. The coordinator executes Algorithm 2; after receiving a report for flow f, if the number of reports received for f exceeds threshold R, the coordinator determines that this mule flow is now an elephant.

Algorithm 2: Coordinator Algorithm

```
Input: Reporting Threshold (R)
Output: Heavy Hitter Set (H)
Func HandleReport(f):
    Reports_f ← Reports_f + 1
    if Reports_f ≥ R
        H ← H ∪ {f}
```

3.1.4 Configuring Parameters. Configuring the parameters (τ, r, and R) to strike a balance between accuracy and communication cost is non-trivial. For example, we want to set τ high enough such that it can effectively differentiate between the mule and mole flows, but low enough that it does not increase the number of flows classified as mules by the switch and, consequently, the communication cost. Previous work [9] demonstrated tight bounds on communication and error by selecting specific values of $r = 1/k$ (k is the number of switches) and τ by introducing an approximation factor (ϵ). Their results show that this approach can achieve high accuracy with modest communication overhead that does not grow proportionally to the number of switches in the network. We generalize the results from prior work by setting $r = 1/k$, $\tau = \epsilon T/k$, for $0 < \epsilon < 1$; the coordinator then determines that a flow is an elephant after receiving $R = kr/\epsilon$ reports. When $r = 1/k$ this threshold simplifies to $R = 1/\epsilon$, but in Section 5 we describe how we might vary the value of r when tuning all of the Herd's parameters together.

Example. Let us consider an example topology with $k = 10$ switches, global threshold $T = 2500$ and we choose an

approximation factor of $\epsilon = 0.1$. In this case, switches would report every $\tau = 25$ packets to the coordinator with probability $r = 0.1$. The coordinator would therefore declare any mule an elephant after receiving $R = 10$ reports. Here, a single τ and r apply to all flows at all switches in the network.

3.2 Locality-aware Reporting Parameters

The protocol we described in the previous section implicitly assumes that all of the k switches in the network are equally likely to observe a portion of the traffic for a given flow. This assumption results in lower local thresholds (τ) as networks grow large. A smaller τ will result in the switch determining that more mole flows are mules, and, ultimately, increase the communication cost. However, in practice, most flows exhibit spatial locality, i.e., only a subset of edge switches observe traffic for a given flow. If a flow is only observed at $l << k$ locations, then we should configure the parameters based on this smaller number of switches. Accounting for this locality would increase τ, which, in turn, ensures that fewer moles are unnecessarily promoted to mules, thus reducing the communication cost.

3.2.1 Configuring Parameters. We now require an additional parameter l_f to account for the spatial locality. Here, l_f denotes the *number* of switches that observe flow f. For now, we can assume that we know the locality parameters for all flows *a priori* because forwarding state can be used to infer this information. Accounting for this locality parameter, we adjust the local threshold as $\tau = \epsilon T / l_f$ and reporting probability as $r = 1/l_f$ for each flow at the switch. The coordinator reports flow f as a heavy hitter when it receives $R = 1/\epsilon$ reports from the switches. Returning to Algorithm 1, we must augment the key-value store D to maintain l_f for each flow. The values for τ and r are then calculated based on looking up $D[f].l$.

Example. Let us return to our example with $k = 10$ switches, global threshold $T = 2500$, and an approximation factor of $\epsilon = 0.1$. Let us now consider that a particular flow f is observed only at $l_f = 2$ switches in the network. We now can increase both our bundle size to $\tau_f = 125$ and reporting probability to $r_f = 0.5$. The coordinator would still declare a mule an elephant after receiving $R = 10$ reports for the flow, but there are now fewer mules sending reports to the coordinator due to the larger bundle size. Now, the threshold (τ) and reporting probability (r) can vary based on how many switches actually observe the flow.

3.2.2 Tracking Spatial Locality. In reality, the locality of flows in a network changes due to routing updates, misconfiguration, and failure; l_f must be tracked dynamically. Thus, Herd introduces a protocol that independently tracks changes in the spatial locality for flows directly in the data plane. At all times, a switch has knowledge of which flows it expects to observe (more on this in Section 4.2). When a switch receives a packet from an unexpected flow, the switch sends a `Hello` message to the coordinator. As shown in Algorithm 3, the coordinator extracts the flow (f) and switch identifier (s) from the `Hello` message and looks up the value of l_f. It also updates a data structure (S_f) that maintains a mapping of flows to switches. Finally, it sends the updated parameter to all switches in S_f. To avoid updating the parameter due to spurious or transient conditions, we choose to update the locality parameter for a flow only when the set of switches actually observing the flow (S_f) is doubled.

Algorithm 3: Coordinator Algorithm for Learning l_f

```
Func HandleHello (hello):
    f, s ← ExtractFlow(hello)
    if s ∉ S_f
        S_f ← S_f ∪ s
        if |S_f| ≥ 2 l_f
            l_f ← |S_f|
            Send(x ∈ S_f, l_f)
        else
            Send(s, l_f)
```

4 SWITCH DATA STRUCTURE

The coordination protocol described in the previous section assumed that switches could store a counter and l_f for each flow. Although modern programmable switches enable flexible packet processing directly in the data plane, the amount of memory available for stateful operations is orders of magnitude smaller than what the coordination protocol described earlier requires. In this section, we again exploit the observations that (1) only a few flows are heavy hitters, and (2) flows exhibit spatial locality to reduce the memory footprint on the switches. We first describe how we avoid maintaining state for the many mice flows, and then how we decouple storing the locality parameter from the flow counters,

4.1 Separating Mice from Moles

To reduce the memory footprint, we need a data structure that can avoid consuming resources for local mice flows, which are too small to significantly contribute to a network-wide elephant. This structure therefore needs to effectively and efficiently separate mole flows from mouse flows.

Why not just use sketches? In order to separate mice from moles, we could use a count-min sketch (CMS) to estimate the size of all flows, and then only allocate an exact counter when a flow exceeds some minimum threshold ($\tau' < \tau$). Conventionally, approximate data structures, such as CMS, have been used to monitor heavy hitters with bounded memory and error. [8, 14, 15, 28] Unfortunately,

these data structures were designed for tracking single-site *elephants*, and are therefore normally used for identifying flows which take up a large portion of the traffic at a single switch. Using a CMS to accurately estimate both *mouse* and *mole* flows instead would require much larger sketches to achieve acceptable accuracy.

For example, a CMS that uses b bits per row with r rows and processes N packets will produce an estimate that errs at most $2N/b$ with probability at least $1 - (1/2)^r$ from the true count. Assuming $100M$ packets are processed by the switch in a monitoring interval, setting $b = 10K$ will result in an error of at most $20K$ and we would need to allocate $b = 100K$ to get an error of at most 2,000. For the task of counting small flows, a CMS does not strike the right balance between state and accuracy.

Sample and Hold the Moles. Although mice flows comprise a large portion of the total flows in a network, these flows are few in total packet count. Therefore, we can use sampling to effectively filter out those flows whose count is less than the inverse of the sampling probability, in expectation. For flows that we do sample, however, we store an exact counter so that we can separate mules from moles as described in Section 3.1.1. While this technique will not prevent *all* mouse flows from erroneously being promoted to moles, it does eliminate enough mouse flows to store the sampled mole flows in the limited switch memory.

Algorithm 4: Sample and Hold Switch Algorithm

```
Func UpdateAndCheck(f, D):
    if f ∈ D
        l_f ← D[f].l
        τ_f ← εT/l_f
        D[f].count ← D[f].count + 1
        if D[f].count ≥ τ_f
            D[f].count ← 0
            return True
    else
        if Flip(s)
            D[f].count ← v
    return False
```

We describe the `UpdateAndCheck` function in Algorithm 4 using a key-value store D of limited size. For each incoming packet belonging to a flow f, the switch first checks if f is currently in the key-value store D. If not, the switch inserts f into D with probability s. If the switch decides to insert the flow, it initializes the count in D to an initial value ($v = \frac{1}{s}$). We discuss how to set sampling rates and initial values in more detail in Section 5. If the flow is in D, the switch first looks up the l_f parameter stored in D to calculate τ_f and r_f.

src	dst	l
10.0.0.0/8	20.0.0.0/8	3
5.0.0.0/8	6.0.0.0/8	2

flow	count
f_1	10

Figure 3: Herd Switch Data Structures for locality.

The switch then updates the count and checks to see if it will report this bundle to the coordinator.

We can reduce the memory footprint of D by using a low sampling probability. However, we want a sampling probability high enough such that when sampled, the initial value does not exceed (τ), therefore automatically promoting all moles to mules. We prevent this by selecting a sampling probability greater than $\frac{1}{\tau}$.

4.2 Locality-aware Data Structure

As we discussed in Section 3.2, a locality-aware coordination protocol ensures that switches can use a higher reporting threshold τ_f based on the locality parameter l_f. However, the forces that affect flow locality (e.g., Internet routing) can operate at a granularity independent of the granularity at which we may want to monitor flows. We must account for this disparity in the locality-aware data structure.

4.2.1 Storing Parameters at the Granularity of Locality. So far, we have assumed that switches can calculate per-flow parameters such as reporting probability (r_f), and reporting threshold (τ_f) based on the locality parameter l_f. The overhead of maintaining these parameters at the flow-level of granularity could outweigh the benefits of locality awareness both in terms of communication and memory costs. Fortunately, the granularity at which flows exhibit spatial locality is much coarser than that required for monitoring. For example, forwarding decisions are usually made at the granularity of source and/or destination IP prefixes, affecting where flows will display locality. On the other hand, network operators might be interested in detecting heavy hitters at the five-tuple or source-destination pair address granularity. We will now show how we leverage this observation to reduce the memory footprint and communication overhead for maintaining locality-aware parameters.

To leverage this observation, we define a group ($g_{src,dst}$) based on source-destination IP prefix pairs, such that $g_{src,dst} = \{f | f.srcIP \in src, f.dstIP \in dst\}$. As shown in Figure 3, rather than maintaining and updating the locality parameter on a per-flow basis, a switch maintains the locality parameter based on the group that displays this locality. We store a group (g) at a switch if at least one flow $f \in g_{src,dst}$ is observed at the switch. Algorithm 1 is modified so that the switch extracts the locality parameter l_g for a flow f based

on the group to which the flow belongs. The switch then calculates the parameters τ_g and r_g and supplies τ_g as an additional parameter to `UpdateAndCheck`. The updated Algorithm 6 shows small changes needed from the original switch algorithm and is provided for reference in Appendix A. Algorithm 3 is also modified slightly such that all variables indexed by f are now indexed by g.

5 TUNING SYSTEM PARAMETERS

So far, we have discussed how we designed the coordination protocol and switch data structure to achieve high accuracy with limited communication and memory costs. However, when configuring the parameters for each (e.g., sampling and reporting probability, local threshold, etc.) in isolation, we may actually choose values for these parameters that worsen performance under resource constraints. Given the operational constraints on switch memory S and the communication overhead C, we now describe an algorithm for determining the optimal parameter configuration such that the system achieves high detection accuracy within the constraints. In this section, we use representative packet traces to first set the sampling rate based on the switch memory bounds (§ 5.1). We then show how to set reporting parameters (§ 5.2) based on bandwidth constraints. Finally, we describe a heuristic algorithm that maximizes the detection accuracy for a given bandwidth and memory budget (§ 5.3) by choosing "good" values of parameters based on fundamental bounds (§ 5.4). In this section, we assume a single flow group for the sake of clarity since the same process can be applied to multiple flow groups.

5.1 Sampling Based on State Constraints

If we had unlimited memory in the switch, we could maintain exact counters for all of the flows by setting the sampling probability to $s = 1$. However, the available memory is finite, and Herd needs to account for this limitation. Given the operational constraints on a switch's memory is S, our goal is to determine the highest sampling probability s that satisfies this constraint given the workload.

If P denotes the number of packets observed by the switch, then we expect the memory usage to be (sP). Therefore, the maximum sampling probability the switch data structure could support is $\frac{S}{P}$ for a given bound on switch memory S. However, if the flow size distribution is known a priori, we can select a higher sampling probability, based on the number of moles that the switch can maintain. The `GetSampling` function in Algorithm 5 shows how we use the given data (D) to empirically determine the highest possible sampling probability, given the memory constraint S. Let M denote the set of mole flows observed at the switch. The `CalculateMoles` function is used to calculate M, given the workload D and sampling probability s; the function iteratively searches for the largest set M that the switch can support.

Symbol	Meaning
	Given
T	Global threshold
C	Communication budget per switch
S	Memory budget per switch (# counters)
k	Total number of ingress switches
l	Number of switches which observe a flow
D	Training Data
	Determine
ϵ	Approximation Factor
τ	Local (Mule) threshold
M	Set of moles observed at switch
U	Set of mules at a switch
r	Reporting probability to coordinator
s	Sampling probability at a switch

Table 1: Network-Wide Heavy Hitter Parameters

To ensure that the actual set of moles sampled at the switch contains true mules, we must ensure that we sample with probability greater than $\frac{1}{\tau}$ in order to ensure that the count of the sampled flows is strictly less than the mule threshold τ, in expectation. In summary, the local mule threshold determines the lower bound for sampling probability, and the available switch memory sets its upper bound.

5.2 Reporting Based on Communication

If we had unlimited bandwidth, we could set ϵ as small as needed to achieve the desired accuracy and incur the resulting communication overhead. However, communication resources are also constrained so we need to adjust the system parameters accordingly. Given the communication bound C, we must configure the reporting probability.

In section 3.1.4, we calculate the reporting probability as $\frac{1}{l}$ to achieve good accuracy and grow to large size networks. However, we must adjust the local threshold (τ), reporting probability (r), and global reporting threshold (R) for high accuracy given communication constraints. The function `DeriveReporting` configures these parameters based on a given value of (ϵ). A switch sends $\frac{T}{\tau}$ reports to the coordinator for each mule flow when $r = 1$. We denote U to be the set of mule flows observed at a switch. The `CalculateMules` function is used to determine U, given the set of moles and the local threshold as input. Finally, the algorithm uses the set of mule flows and the communication budget to calculate the reporting probability. The total number of reports sent to the coordinator is bound by the total number $\frac{T|U|r}{\tau}$, in

expectation. For a given C, the upper bound on reporting probability is, therefore, $\frac{C \cdot \tau}{T|U|}$.

5.3 Tuning for High Accuracy

Once we have set the sampling rate and determined the reporting probability, we can now find an optimal local threshold. Since the local threshold can be tuned with the approximation factor ϵ, we describe an algorithm that searches for an optimal value based on the given parameters of the system. In the TuneAccuracy function, the algorithm uses representative packet traces to empirically compute the moles and mules and set the parameters of the system as described above. After calculating all parameters, the algorithm calls the GetAccuracy function to determine the accuracy of the System using this parameter configuration. The algorithm then iteratively searches for a value of ϵ that is at least ϵ_{min} where the accuracy of a succeeding iteration is less than the preceding iteration and then terminates.

5.4 Selecting the Right Values of ϵ

Many counter estimation techniques often leave selecting the approximation factor (ϵ) to the user. However, näively choosing values of ϵ without regard to the other parameters can actually result in poor system performance. By wisely selecting values of ϵ, we can both reduce the range of possible values for ϵ that Algorithm 5 has to explore and avoid the detrimental effects that can occur when discretizing parameters. For example, although ϵ is a real number, it is used to calculate integer thresholds for the local switches (τ) and the global number of reports (R). When determining these discrete values based on a continuous calculation, rounding can significantly degrade system performance, which we describe and show later in Section 7.2. To reduce this error, we can select values of ϵ that result in whole integer values for other parameters. First, we begin by asserting that there are maximum (ϵ_{max}) and minimum values (ϵ_{min}) that we wish to explore shown here in Theorems 1, 2. We include the proof of each theorem below in Appendix B.

Theorem 1 (ϵ_{min}). *For all k, l, T where $k \geq l$ and $\tau \geq 1$, there exists an $\epsilon_{min} \geq \frac{k}{T}$.*

Since we also know that as the local threshold on the switch grows too large, the accuracy of the system degrades. Consequently, we can ignore values of ϵ that would result in local thresholds larger than (T/k), shown here in Theorem 2.

Theorem 2 (ϵ_{max}). *For all k, l, T where $k \geq l$ and $\tau \leq \frac{T}{k}$, there exists an $\epsilon_{max} \leq \frac{l}{k}$.*

Intuitively, we also know that ϵ_{max} should be greater than or equal to ϵ_{min}. Whenever that condition does not hold, we cannot use our algorithm to find the best value of ϵ. We can use Theorems 1 and 2 to determine in what combination of *compatible parameters* this condition will hold.

Algorithm 5: Algorithm for tuning parameters.

```
Func GetSampling(S, D, mole_tau):
    s ← 1 / mole_tau
    M ← CalculateMoles(D, s)        // Section 5.1
    while |M| < S do
        mole_tau ← mole_tau − 1
        M ← CalculateMoles(D, s)
    end
    return mole_tau

Func DeriveReporting(C, ε, l, s):
    τ ← εT / l                      // Section 5.2
    M ← CalculateMoles(D, s)
    U ← CalculateMules(M, τ)
    r ← C·τ / T|U|
    R ← l·r / T
    return R, U, r, τ

Func TuneAccuracy(T, S, C, D, l):
    A_max ← 0                       // Section 5.3
    mole_tau ← GetSampling(S, D, T)
    s ← 1 / mole_tau
    while ε ∈ [ε_max ... ε_min] do
        R, U, r, τ ← DeriveReporting(C, ε, l, s)
        A ← GetAccuracy(D, R, T, U, r, s, τ,)
        if A ≥ A_max
            ε_max ← ε
            ε ← ε − σ               // Section 5.4
            A_max ← A
        else
            break
    end
```

Theorem 3 (Parameter Compatibility). *By Theorems 1 and 2, for all k, l, T where $k \geq l$, parameters are compatible when $T \geq \frac{k^2}{l}$.*

To minimize quantization error when selecting values of ϵ, we should seek to ensure that the calculated local threshold is a whole integer without rounding. We therefore select a *quantization step* ($\sigma = \frac{l}{T}$) and set ϵ as an integer factor of this step to ensure that all values of τ are whole integers.

Theorem 4 (Quantization Factor (σ)). *For all k, l, T, ϵ, τ where $\sigma = \frac{l}{T}$, there exists an integer factor (n) that ensures τ is a whole integer.*

Finally, we can use Theorems 1, 2 and 4 to determine the values of $n_{min} \geq \frac{k}{l}$ and $n_{max} \leq \frac{T}{k}$.

6 P4 PROTOTYPE

We now describe our P4 prototype that implements the coordination protocol and the switch data structure. A modern PISA [24] switch consists of a programmable parser that can extract user defined fields from a packet. It stores the extracted packet fields, as well as additional metadata fields in a packet header vector (PHV). A multi-staged packet processing pipeline transforms this PHV using a fixed amount of resources in each stage. Each stage consists of one or more match-action tables (MAT) that consume the finite TCAM, SRAM, and ALUs to match on and transform fields in the PHV. At the end of the processing pipeline, a deparser serializes the PHV, into a packet before sending it to an egress interface. In this section, we describe the overall structure of our P4 prototype and then describe the challenges that we faced while implementing our prototype in this architecture.

6.1 The Life of a Packet

When a packet enters the PISA pipeline, we first determine to which group ($g_{src,dst}$) the packet belongs. The group identifier is then used to match a rule in a MAT (§ 6.2), in order to determine the local threshold (τ_g), and reporting probability (r_g) for that group. Next, two independent but biased coin flips (§ 6.3) are performed; the first coin flip is based on the sampling probability (s) and the second based on the reporting probability (r). Both results are stored as packet metadata values $flip_1$ and $flip_2$, respectively, for use later in the pipeline. Finally, we must check if the flow is stored in the hash tables that implement the key-value store D (§ 6.4). If the flow is found, its counter is incremented. If the counter is greater than τ_g, the counter is reset to 0; if $flip_2$ == `true` as well, a report is then sent for this flow. If the flow was not found and $flip_1$ == `true`, then the flow is sampled and stored in the hash table. If no empty space is found in the hash table, then the packets in the flow are sent to the coordinator—trading communication cost for accuracy.

6.2 Storing Locality Parameters with MAT

So far, our presentation of the switch data structure assumed that the switch could compute the local threshold (τ_g), and reporting probability (r_g) by itself if it knew l_g. However in practice, such computations require floating point arithmetic that is currently not supported in PISA switches. Since both parameters are specific to each locality group, we can store both parameters in a match-action table as shown in Figure 3. However, instead of storing l_g itself, we can precompute the values of (τ_g) and (r_g) and store them instead. Consequently, we can retrieve these values using a single lookup in the match-action table where τ_g and r_g are stored. This example shows how seemingly simple algorithms must be altered in order implement them on (today's) PISA switches.

6.3 Flipping Coins with Hashes

The `Flip` function, used in Algorithms 2, 4, and 6, requires flipping a biased coin in the switch. A naïve implementation of `Flip` also requires support for floating point arithmetic on the switch. Instead, we represent floating point values ($0 < i \leq 1.0$) as unsigned integers, which is similar to how other works [1] have implemented probabilistic techniques with PISA switches. We then use a combination of a packet's timestamp and other header fields to compute a 32-bit hash value. A `Flip` operation returns `True` if the computed hash is less than $\lceil 2^{32} i \rceil$. This approach introduces a small quantization error since we can only represent probabilities as multiples of $\frac{1}{2^{32}}$.

6.4 Key-Value Store with Hash Tables

The `UpdateAndCheck` function (detailed in Algorithm 4) for updating the counters of mole flows requires implementing the key-value data store D. We can implement this data store as a hash-indexed register array within a single stage. However, these hash-indexed arrays will likely encounter collisions in a single stage. To address this problem, we implement D as a multi-stage hash-table. When inserting a value into D, we insert the value in the first hash table that contains no collision. Though this approach ensures that a switch can maintain counters for a large number of mole flows, the limited memory per stage and number of stages ensures that collisions are inevitable as the number of flows grows large.

7 EVALUATION

In this section, we quantify how Herd makes efficient use of limited communication and state resources to detect network-wide heavy hitters with as high accuracy as possible. We use real-world packet traces to demonstrate how combining probabilistic counting with probabilistic reporting reduces Herd's memory footprint by 38% and bandwidth footprint by 17% to report network-wide heavy hitters with 97% accuracy.

7.1 Setup

To quantify Herd's performance, we run a simple network-wide heavy-hitter query to determine which flows (based on the standard five-tuple of source/destination IP address, source/destination port, and transport protocol) send a number of packets greater than a global threshold (T) during a rolling time window (W).

Simulation experiments. For our experiments, we monitor at the edge switches of the network where the number of edge switches (k) is 10 — representative of a wide-area network connecting multiple data centers for cloud providers [13]. For all experiments, each flow shows affinity for two ingress switches, i.e., $l = 2$, based on the source IP

Technique	Prob. Counting	Prob. Reporting	State Required
Strawman	✗	✗	345K
RLA	✗	✓	345K
Sampling	✗	✓	N/A
Herd	✓	✓	211K

Table 2: Comparing to other Heavy-Hitter detection techniques. Herd uses both probabilistic counting and reporting where other approaches use only one.

address. We choose a global threshold that corresponds to the 99.99^{th} percentile flow count in the packet trace.

Packet traces. To emulate real-world traffic distributions, we used CAIDA's anonymized Internet traces from 2016 [19]. These traces consist of all the traffic traversing a single OC-192 link between Seattle and Chicago within a major ISP's backbone network. Each minute of the trace consists of approximately 64 million packets. For our experiments, we use a time window (W) of five seconds resulting in around 5 million packets and hence $\approx$ 270K unique flows per window.

Since the packet traces are collected from a single link only, we associate packets from the trace with a given ingress switch based on a hash of the source IP address. For each source IP address, we assign an affinity for a specific ingress switch with probability p. Packets from a given source IP are, therefore, processed at a "preferred" switch with probability p and at the other switches with probability $(1-p)$. For the case of $l = 2$, this distribution simulates a primary/alternate relationship on ingress for a single source and $p = 0.95$.

7.2 Baseline Herd Performance

To demonstrate the benefits of combining probabilistic counting and reporting, we quantify the amount of state and communication overhead for Herd and compare it with the existing heavy-hitter detection techniques that either employ probabilistic counting or reporting, but not both.

Alternative Approaches. First, we consider a strawman solution that makes use of neither probabilistic counting or reporting; each switch maintains counters for every flow (all flows are moles) and reports all the counters to a central coordinator at the end of a window. Second, we consider a solution based on the randomized reporting technique [9]; where the switch still treats all flows as moles, but it probabilistically reports mules to the coordinator with parameters that ignore locality. Finally, we consider a solution based on packet sampling technique [18], which probabilistically samples packets based on a sampling rate and reports all of those samples to the coordinator.

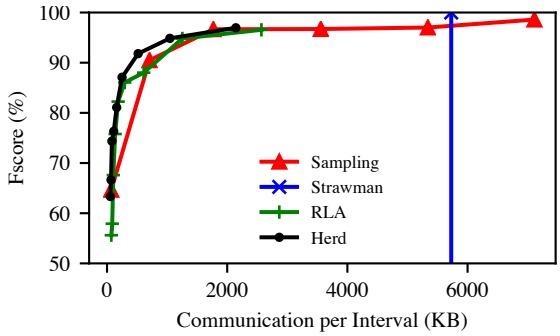


Figure 4: Communication vs. Accuracy. Herd can achieve accurate results with comparably lower communication overhead than existing approaches.

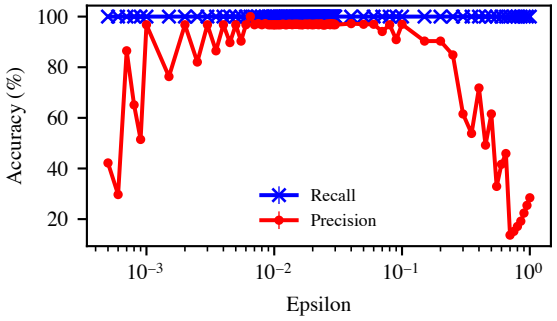
Communication and State Savings. We quantify the state overhead as the number of stateful counters required at the switch and the communication overhead as kilobytes sent to the coordinator for each window interval. We quantify accuracy in terms of both precision (PR) and recall (RE) and present them as a single F_1 score calculated as $\frac{2\times PR\times RE}{PR+RE}$. In Table 2, we see that Herd achieves 38% savings in the state required for alternate approaches. In Figure 4, we compare how much communication is needed to reach a certain level of accuracy. We see that to achieve an F_1 score of 97%, Herd communicates 17% less than sampling packets with a probability of 0.075.

As the threshold determining heavy-hitters decreases, this advantage becomes more pronounced. We evaluated the communication/accuracy tradeoff compared with sampling for three different heavy-hitter thresholds ranging from the 99.99^{th} to the 99^{th} percentile thresholds. Herd performs strictly better than the sampling approach in all cases, except in the 99.99^{th} percentile threshold where the sampling probability is greater than 0.05 – an unrealistically-high sampling probability for modern data centers. Graphs showing the affect of different thresholds is shown in Appendix C.

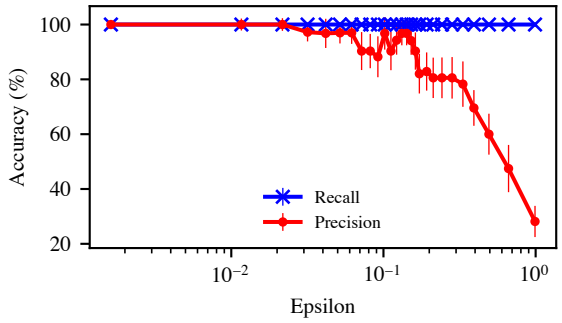
7.3 Tuning for Resource Constraints

So far, we have demonstrated that combining probabilistic counting with reporting reduces both the memory and bandwidth footprint for detecting network-wide heavy hitters. We will now show the relationship between Herd's performance (accuracy) and configuration parameters (ϵ) for different operational constraints. These relationships guide the design of our tuning algorithm.

Unconstrained Performance. We first show the relationship between accuracy and configuration parameters (derived from ϵ) for the unconstrained case. Figure 5, shows both precision and recall for Herd while varying ϵ without any resource constraints. As we choose a smaller epsilon, the accuracy of the results generally improves. However, we

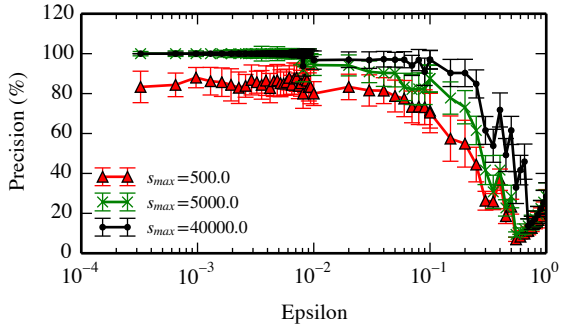


(a) Un-quantized Values of ϵ

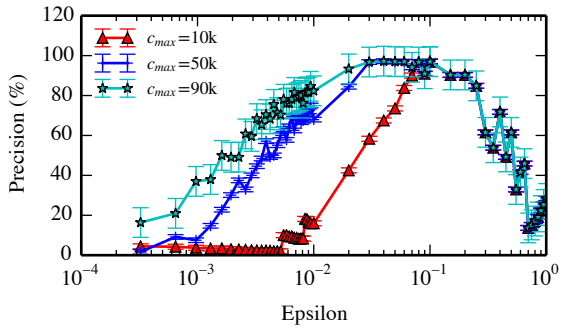


(b) Properly Quantized Values of ϵ

Figure 5: Accuracy while varying values of ϵ.



(a) Constrained in state unconstrained by communication



(b) Constrained in communication unconstrained by state

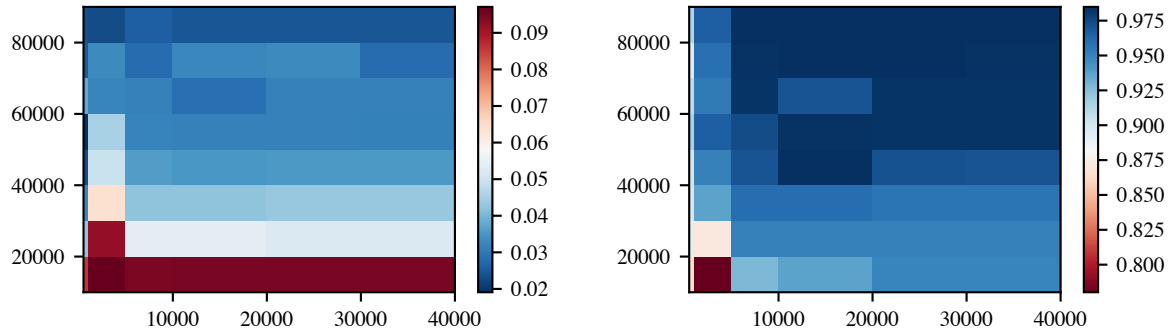
Figure 6: Precision for various ϵ with constraints.

do see that this relationship is not strictly monotonic. We observe the bands of decreasing precision when ϵ is very large or very small. These bands are caused by the the quantization errors introduced when discretizing system parameters such as τ and R (discussed in Section 5.4). For example, in the bands of decreasing precision on the right of Figure 5(a), each data point corresponds to a single global reporting threshold (R) but a range of local thresholds (τ) that can vary by up to a thousand. On the left side of the graph, the opposite is true; a single local threshold corresponds to a range of global reporting thresholds. By properly quantizing values of epsilon as shown in Section 5.4, we do not observe these artifacts as shown in Figure 5(b).

Constrained State. Here, we limit the number of counters each switch can store. By choosing the sample-and-hold technique for storing counters at switches, we expect that for a given state capacity (S) and sampling probability (s), the data structure will contain a mixture of both small and heavy flows. However, as we increase S and s, we will count more *small flows* in expectation. As shown in Figure 6(a), increasing S and s improves the precision of the results, but the improvement diminishes as S grows large.

Constrained Communication. We expect that as we decrease ϵ, we should increase communication and increase the accuracy of the results. Algorithm 5, shows us how to adjust the reporting probability based on the available communication capacity (C), however, as we decrease the reporting probability and the reporting threshold to cope with the communication bound, too many false positives are generated as shown in Figure 6(b). This trend is not reflected in the unconstrained case. These results show that the relationship between accuracy and ϵ is non-monotonic, though communication and state costs do monotonically decrease as ϵ increases. These observations guided the design of our tuning algorithm that empirically tries to find the largest value of ϵ that achieves the highest detection accuracy.

Tuning Efficacy. To determine the effectiveness of our tuning algorithm, we varied both state and communication constraints and let Algorithm 5 find the best value of ϵ. In Figure 7(a), we see the performance of tuning when both state and communication constraints are imposed, shown on the x and y-axes, respectively. We see that as resource constraints are relaxed, the system finds a smaller value of ϵ. In Figure 7(b), we see the system's accuracy under the same constraints using the best value of ϵ determined by tuning ϵ for those constraints. Herd is able to produce more accurate results as the constraints are relaxed, but Herd also provides



(a) The best value of ϵ after tuning.

(b) Accuracy with best epsilon.

Figure 7: Efficacy of tuning.

good accuracy even under strict memory and bandwidth constraints.

8 RELATED WORK

Scalable measurement techniques. NetFlow [6] was the first standardized approach for collecting ongoing telemetry from network switches. However, Netflow incurs significant CPU overhead or specialized hardware to run efficiently. Packet sampling [18] emerged as the de facto technique to cope with both the memory and communication limitations. However, packet sampling can introduce significant inaccuracy to detecting heavy hitters, especially on small time scales [18]. FlowRadar [14] reduces the memory and communication overhead using a novel encoding of flow counters. Similarly, CSamp [22] provides a sampling mechanism for network-wide measurements. While both of these works are general-purpose solutions for performing network-wide measurement of most flows, we offer a tailored solution for continuous, network-wide monitoring of a global threshold distributed across several switches.

Single-switch heavy hitters with limited state. Prior work showed how to use compact data structures (e.g., count-min sketch [8] and Space-Saving [16]) to compute heavy hitters on a single switch. However, Sivaraman et al. [23] showed that implementing such algorithms with state-of-the-art programmable switches is difficult. Similarly, other techniques such as Cuckoo [17] and d-left [25] hashing can detect heavy hitters with small state, but they prove to be impossible or impractical to implement in modern programmable switches [4]. Recent work, such as ElasticSketch [26], offers a technique to avoid maintaining state for mouse flows in the data plane by offloading the computation to the control plane. Other solutions, such as SketchVisor [12], rely on *software* packet processing which limits their abilities to handle very high data rates. Our approach, based on the sample-and-hold [10] technique, uses sampling to filter out mice flows completely in the data plane and only maintains per-flow state for potential heavy hitters.

Network-wide heavy hitters with limited communication. Detecting network-wide heavy hitters is an instance of the continuous distributed monitoring (CDM) problem [7]. This formulation of the problem has enabled theoretical analyses that demonstrated upper and lower bounds on the communication complexity [9, 27] for both deterministic and randomized solutions. In our work, we extend the basic model from these theoretical works to account for the realities of flow affinity in modern networks [20, 21], as well as the capabilities of programmable switches to support these protocols. Recent work [11] showed that using adaptive local thresholds to account for flow locality could reduce communication overhead for computing network-wide heavy hitters exactly, but that solution does not scale as the number of nodes increases and requires much more communication overhead than our approach. In contrast, our work accounts for flow locality in the CDM model and the communication costs do not scale in proportion to the number of switches in the network. Our solution also offers tunable accuracy based on bandwidth constraints.

9 CONCLUSION

We presented a system for detecting network-wide heavy hitters with high accuracy under communication and state constraints. We combined sample-and-hold counting on the switches with probabilistic reporting to a central coordinator. Based on these reports, the coordinator adapts the parameters at each switch to the spatial locality of the flows. We presented an algorithm for tuning the various System parameters to increase detection accuracy under resource constraints. In the future, we would like to apply more robust techniques to avoid overfitting system parameters to the training data.

REFERENCES

[1] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE ICNP*.

[2] Ran Ben-Basat, Gil Einziger, Shir Landau Feibish, Jalil Moraney, and Danny Raz. 2018. Network-wide routing-oblivious heavy hitters. In *ACM/IEEE ANCS*.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review* (July 2014).

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*.

[5] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. 2009. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *ACM SIGCOMM Workshop on Research on Enterprise Networking*.

[6] Benoit Claise. 2004. Cisco Systems NetFlow Services Export Version 9. *RFC 3954* (2004).

[7] Graham Cormode. 2011. Continuous Distributed Monitoring: A Short Survey. In *International Workshop on Algorithms and Models for Distributed Event Processing*.

[8] Graham Cormode and Shan Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms* (2005).

[9] Graham Cormode, S Muthukrishnan, and Ke Yi. 2011. Algorithms for Distributed Functional Monitoring. *ACM Transactions on Algorithms* (2011).

[10] Cristian Estan and George Varghese. 2003. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems* (2003).

[11] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *ACM SIGCOMM Symposium on SDN Research*.

[12] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *ACM SIGCOMM*.

[13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined WAN. In *ACM SIGCOMM*.

[14] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*.

[15] 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 *ACM SIGCOMM*.

[16] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2006. An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Transactions on Database Systems* (2006).

[17] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* (2004).

[18] P. Phaal, S. Panchen, and N. McKee. 2001. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. *RFC 3176* (2001).

[19] report [n. d.]. The CAIDA Anonymized Internet Traces 2016 Dataset. https://www.caida.org/data/passive/passive_2016_dataset.xml. ([n. d.]).

[20] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*.

[21] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *ACM SIGCOMM*.

[22] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. 2008. cSamp: A System for Network-Wide Flow Monitoring. In *USENIX NSDI*.

[23] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SIGCOMM Symposium on SDN Research*.

[24] url [n. d.]. Barefoot's Tofino. https://www.barefootnetworks.com/technology/. ([n. d.]).

[25] Berthold Vöcking. 1999. How Asymmetry Helps Load Balancing. In *IEEE Symposium on Foundations of Computer Science*.

[26] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *ACM SIGCOMM*.

[27] Ke Yi and Qin Zhang. 2009. Optimal Tracking of Distributed Heavy Hitters and Quantiles. In *ACM SIGMOD-SIGART-SIGACT Symposium on Principles of Database Systems*.

[28] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*.

A MODIFIED SWITCH ALGORITHM

As described in Section 4.2, Algorithm 1 is modified so that the switch extracts the locality parameter l_g for a flow f based on the group to which the flow belongs, and then calculates the parameters τ_g and r_g. Algorithm 6 shown below details the needed modifications.

Algorithm 6: Switch Algorithm

Input: See Table 1
Func `ProcessPacket(`pkt`)`:
  $f \leftarrow$ `ExtractFlow(`pkt`)`
  $l_g \leftarrow$ `GetLFromGroup(`f`)`
  $\tau_g, r_g \leftarrow \tau = \epsilon T / l_g, \frac{1}{l_g}$
  $exceeds \leftarrow$ `UpdateAndCheck(`f,D,τ_g`)`
  if $exceeds$
    if `Flip(`r_g`)`
      `Report(`f`)`

B PARAMETER SELECTION PROOFS

We Provide proofs for theorems from Section 5.4.

THEOREM 1 (ϵ_{min}). *For all k, l, T where $k \geq l$ and $\tau \geq 1$, there exists an $\epsilon_{min} \geq \frac{k}{T}$.*

PROOF.

$$\frac{T\epsilon}{max(k,l)} = \tau$$

$$\frac{T\epsilon_{min}}{max(k,l)} \geq 1$$

$$\frac{T\epsilon_{min}}{k} \geq 1$$
$$\epsilon_{min} \geq \frac{k}{T}$$

□

Theorem 2 (ϵ_{max}). *For all k, l, T, where $k \geq l$ and $\tau \leq \frac{T}{k}$, there exists an $\epsilon_{max} \leq \frac{l}{k}$.*

Proof.

$$\frac{T\epsilon_{max}}{min(k,l)} \leq \frac{T}{k}$$
$$\frac{\epsilon_{max}}{l} \leq \frac{1}{k}$$
$$\epsilon_{max} \leq \frac{l}{k}$$

□

Theorem 3 (Parameter Compatibility). *By Theorems 1 and 2, for all k, l, T where $k \geq l$, parameters are compatible when $\epsilon_{max} \geq \epsilon_{min}$.*

Proof.

$$\frac{l}{k} \geq \epsilon_{max} \geq \epsilon_{min} \geq \frac{k}{T}$$
$$\frac{l}{k} \geq \frac{k}{T}$$
$$1 \geq \frac{k^2}{T \cdot l}$$
$$T \geq \frac{k^2}{l}$$

□

Theorem 4 (Quantization Factor (σ)). *For all k, l, T, ϵ, τ, where $\sigma = \frac{l}{T}$ there exists an integer factor (n) that ensures τ is a whole integer.*

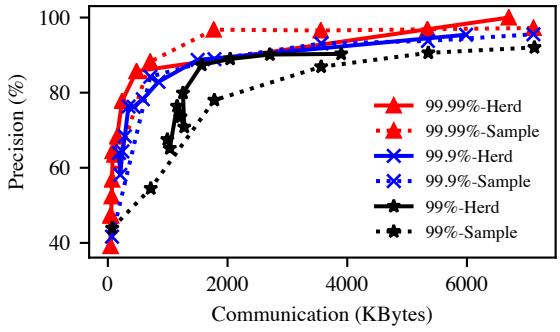
Proof.

$$\epsilon = \sigma n$$
$$\tau = \frac{T\epsilon}{l}$$
$$\tau = n\frac{T\sigma}{l}$$
$$\tau = n\frac{T}{l} \cdot \frac{l}{T}$$
$$\tau = n$$

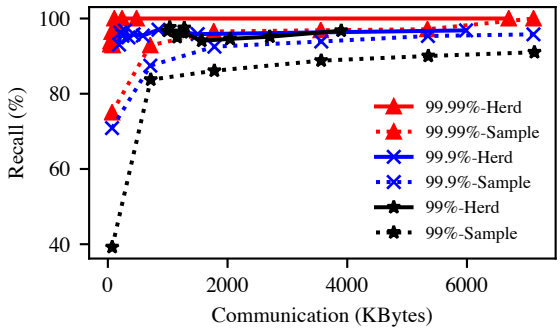
□

C ADDITIONAL EVALUATIONS

Sensitivity to Heavy-Hitter Threshold. In section 7, we showed in Figure 4, that Herd can achieve higher accuracy for less bandwidth compared to existing approaches. In Figure 8, we show the communication/accuracy tradeoff compared with sampling for three different heavy-hitter thresholds ranging from the 99.99^{th} percentile to the 99^{th} percentile threshold. In each case, we see that Herd performs strictly better than the sampling approach except in the 99.99^{th} percentile threshold and the sampling probability is greater than 0.05 which is unrealistically-high for modern data centers.



(a) Precision



(b) Recall

Figure 8: Accuracy vs. Communication Cost and Threshold.