# Classification-assisted Query Processing for Network Telemetry

Gioacchino Tangari
University College London
gioacchino.tangari.14@ucl.ac.uk

Marinos Charalambides
University College London
marinos.charalambides@ucl.ac.uk

Daphne Tuncer
Imperial College London
d.tuncer@imperial.ac.uk

George Pavlou
University College London
george.pavlou@ucl.ac.uk

***Abstract*—Recent network telemetry systems are designed to provide information for securing the network and supporting network reconfigurations. These systems operate as query processors that respond in real-time to a wide range of monitoring queries concerning events, such as traffic anomalies and potential attacks. While existing measurement techniques can extract all the raw data required for query reporting, a significant amount of processing, in terms of data aggregation and evaluation, is involved. This paper proposes a generalized approach for reduced-cost processing of monitoring queries inspired by machine learning workflows. Our solution entails fast classifications trained over recent traffic and applied to sampled subsets of raw measurement data. The classifiers are automatically tuned to match accuracy requirements of query responses and can achieve considerable processing reductions while keeping the accuracy above 98%, as demonstrated by representative query examples.**

## I. Introduction

Network telemetry is vital for network management and security. By extracting and processing a wide range of network metrics it enables knowledge to be generated, which can be used for various tasks including anomaly detection [20] (e.g., root cause analysis [4]) and traffic engineering [16] (e.g., analysis of dynamic traffic demand). Modern network telemetry systems [23], [9], [12] follow the principles of *software-defined measurements* [22], [21], [15]. They provide a high-level interface that hides measurement details to register monitoring requirements, and automatically configure measurement operations to satisfy hardware constraints. In general, they operate as query processors that receive declarative monitoring commands or *queries*, extract raw measurement data from the traffic streams, and process the data to identify a variety of network events and report them in query responses.

Recent research has empowered network telemetry with the right tools to extract raw measurement data from packet streams at line rate. Hashing techniques and *sketch-based* approaches [6], [7], [21], [10] have emerged as standard ways to collect measurement data with a limited computation and memory footprint. While these advances have contributed to the development of efficient extraction mechanisms, the task of *processing* the collected data remains a costly operation. As networks move to larger speed and scale, it becomes crucial to reduce this cost in order to support large numbers of queries and massive traffic volumes.

978-1-7281-2700-2/19/$31.00 2019 

The main methods proposed in the literature for improving data processing efficiency have mostly been focusing on ad-hoc design optimizations, specific to systems/ implementations [16], [9], [18]. In contrast to previous work, this paper presents a novel *generalized* approach to reduce the cost of query processing by intelligently filtering the raw measurement data collected through the monitoring pipeline. We use a methodology based on *fast classifications* to infer query results from small measurement subsets, and take decisions on the portions of data that can be "proactively" filtered prior to the execution of standard data processing operations. In particular, the proposed approach uses lightweight classifiers that *learn* traffic properties based on recent measurements. To protect query responses from classification errors, the classifiers are automatically configured to discard decisions with low confidence levels, according to user requirements on query results accuracy.

To demonstrate its capabilities, we integrate our solution to a state-of-the art software packet-processing pipeline [16], [1] and experiment with representative query examples. Our evaluation results, obtained with a 30-minute traffic trace and a short 10-secs *training* set, show that large fractions of the extracted measurement data – more than $50\%$ and even up to $90\%$ in some cases – can be filtered out while satisfying accuracy requirements above $98\%$, leading to processing time reduction by up to a factor of 10.

## II. Query-based network telemetry

Network telemetry systems must satisfy two fundamental requirements: (i) provide a generic, *declarative* interface, based on the definition of monitoring *queries*, and (ii) cope with the complexity of query processing, from compilation to the final push of monitoring reports, while satisfying stringent monitoring accuracy goals.

### A. Monitoring queries

Monitoring queries are declarative commands issued to identify a variety of events related to the network performance and security. From a logical perspective, they entail combinations of three building blocks. The first is the *match & extract* component, which selects the portion of the network traffic being in the scope of the query, and extracts raw information based on packet size, header fields, or timestamp. The second one, *evaluate*, corresponds to the set of logical and arithmetic

Table I: Representative monitoring queries

| Query | Description | Processing workflow |
|---|---|---|
| Heavy hitters [16] | A traffic aggregate (srcIP) exceeding volume $K_{hh}$ | **extract** 5-tuple bytes, srcIP; **aggregate** on srcIP (*sum* bytes); **evaluate** sum $> K_{hh}$ |
| DDoS attack [21] | A host (dstIP) reached by more than $K_{ddos}$ unique sources | **extract** 5-tuple srcIP, dstIP; **aggregate** on dstIP (*count distinct* srcIPs); **evaluate** count $> K_{ddos}$ |
| Slowloris attack [9] | A host (srcIP) opening more than $K_{sl}$ connections, with average rate below $B_{sl}$ | **extract** 5-tuple bytes, srcIP; **aggregate** on srcIP (*mean* bytes, *count*); **evaluate** mean $< B_{sl}$ & count $> K_{sl}$ |
| Bursty flow source [16] | A host (srcIP) generating more than $X\%$ *bursty* connections, *i.e.,* connections with more than $Y\%$ packets coming in bursts | **extract** 5-tuple, #pkts, #pkts-in-burst, srcIP; **aggregate** on srcIP (*isBursty()*); **evaluate** $\%$(isBursty() $== 1) > X\%$ |

operations that check extracted information against a set of conditions (query predicates). The last component, *aggregate*, is the state aggregation function (e.g., sum, mean, count, stddev) applied to monitoring data for evaluation or reporting purposes. These components can be used to build complex query-processing workflows, and *aggregate-evaluate* functions can be iterated to check different predicates at different levels of data aggregation.

Table I reports the representative monitoring queries used in this paper. Raw monitoring data is extracted and stored for each 5-tuple flow as in [16].

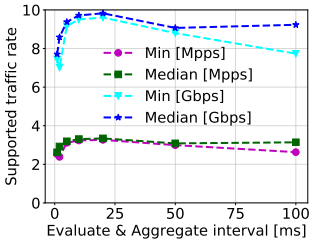



Figure 1: Effects of tweaking the reporting frequency

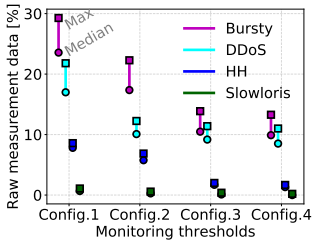



Figure 2: % of raw data *retained* in query responses

## B. Query processing

With networks evolving to larger scale and speed, the challenge for telemetry systems is to handle massive amounts of queries (e.g., 4K on a 10ms time window [16]), while facing both larger numbers of concurrent flows (1000+ on 10ms intervals [19]) and increasing packet rates (10Gbps+ on a single processor core in software-packet processing platforms [16], [10]). As a result, a vast amount of information needs to be processed to build query responses, especially for stateful monitoring [18], [16] (e.g., *groupby*-like state aggregation), with significant computational resources consumed for *aggregate* and *evaluate* execution.

Recent research has investigated how to efficiently match-on-traffic and extract raw measurement data from a packet stream at line rate and with reduced memory footprints. To this end, *sketch*-based techniques [21], [14], [10], [11], *heap*-based solutions (e.g., top-k [13] counting), or simple hash tables [16], [3] can be adopted. However, it is still unclear how to curb the cost of query processing when looking at the *aggregation* and *evaluation* of such amounts of measurement data. Ad-hoc design optimizations have been proposed to improve efficiency, e.g., [9], [18], [16], but these are mainly tailored to specific systems/implementations. Sonata [9] splits *aggregate-evaluate* workload between programmable switches and telemetry streaming collectors at servers. Marple [18] relies on a memory backend on commodity hardware to facilitate stateful *group-by* operations. Trumpet [16] adopts double-buffering of measurement data to interleave the *aggregate-evaluate* steps on previous data with *match & extract* on new traffic.

Generally speaking, different approaches can be explored to reduce the cost of query processing [5]. One possibility is to reduce the pace at which raw data are processed for reporting the query results. However, performing *aggregate-evaluate* operations less frequently leads to increasing accumulation of measurement data over longer periods of time, which results in no substantial benefit. In Fig. 1, we test the impact of relaxed query-results reporting on a software packet-processing pipeline based on Trumpet [1], using a single CPU core[1]. As input, we use a 10min CAIDA trace [2] and apply the queries of Table I. To quantify the query processing cost, we analyze how it affects the monitoring performance in terms of supported traffic rate. As shown in Fig. 1, increasing the reporting period from $1ms$ to $10ms$ (Trumpet default) and $20ms$ improves the maximum traffic rate by approx. $20\%$ and $25\%$, respectively. Going beyond $20ms$, the performance descends as more raw data accumulates prior to processing – more CPU cycles for iteration and memory access.

An alternative approach is to filter the original (raw) information sets considered in the *evaluate* and *aggregate* steps. This is promising when only fractions of the raw measurement data are *retained* to craft query responses. In fact, this applies to all queries issued to detect events concerning unusual behaviors or specific traffic patterns [16], [9], [23]. Fig. 2 shows the percentage of raw measurement data being used for query responses on the same setup of Fig. 1. To make query results more or less selective with respect to total extracted measurement data, we vary the thresholds of Table I[2]. As observed, despite differences between queries, large fractions of raw measurement data ($65\%$+) could always be scrapped

[1]We use a 3GHz CPU, with 8MB shared L3 cache

[2]We select 4 threshold configurations between the ranges $K_{hh} \in [1KB, 100KB]$, $K_{ddos} \in [10, 100]$, $X_{bursty} \in [10, 100]$, $K_{sl} \in [5, 50]$

without influencing the query responses, with peaks above $90\%$ for the most selective configurations.

As the example shows, there is potential in reducing the cost of query processing by filtering the raw measurement data before the *aggregate* and *evaluate* steps are executed. However, to exploit this a few key challenges need to be addressed. First, a solution is required, applicable to a wide range of monitoring queries, to accurately predict portions of the query results, thus enabling intelligent filtering of the raw data. Second, this should guarantee significant processing cost reductions with respect to standard *aggregate* and *evaluate*. Finally, this should not compromise the validity of query responses in terms of query results accuracy. We address these challenges with a novel, *classification-assisted*, query processing approach.

## III. Classification-assisted query processing

To achieve intelligent filtering on the raw measurement data, we enhance the standard query processing workflow with a *fast classification* functionality, whose goal is to reduce the volume of data used as input to the *aggregate* and *evaluate* steps. Our approach is based on the ability to derive classifiers for each query type in order to predict query results from subsets of the raw measurement data. In particular, classifiers are constructed by *learning* traffic properties based on recent measurements.

### A. Overview

An overview of the approach is shown in Fig. 3. The core of our solution is a set of classifiers, one for each query, which take samples of the raw measurement data as input and generate "decisions" on query-related events. For example, for a *Heavy Hitters* query, the classifier takes a sample containing the byte counts of few 5-tuple flows with the same srcIP $x$, and provides a probabilistic answer to the predicate "is $x$ a heavy hitter?". This is achieved by performing *training* based on recent monitoring results, *i.e.,* using labelled samples (for which ground-truth is known) extracted from recent measurement data. The output of classifiers is the key to exclude portions of the raw measurement data from standard *aggregate* and *evaluate* processing. For instance, answering predicate "is $x$ a heavy hitter?" allows raw counters matching srcIP=x to be discarded (*i.e.,* to filter data out), and in case of positive output to "proactively" add $x$ to the query response.

The design of the classifiers follows two main principles. First, classifiers are built using "lightweight", *i.e.,* computationally inexpensive, classification functions. These relies on a logistic regression model, whose run-time execution mainly consists in computing a single $exp$ function. This ensures that the benefits obtained by raw data filtering is not undermined by the computational cost of performing classifications. Second, classifiers are automatically configured to match user-specified requirements, specific to each query, based on the accuracy of query results. Since classification functions using samples are not error-free by definition, this ensures that query responses are protected from erroneous results. Based on such configuration (referred to here as classifier *validation*), part of the classifier outputs can be disregarded when the estimated error

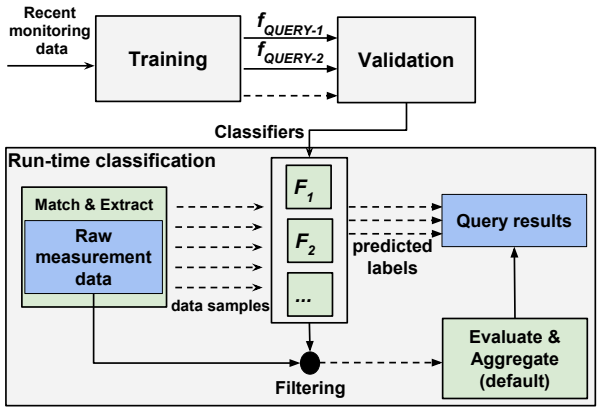



Figure 3: Classification-assisted query processing

probability is not negligible. More specifically, each classifier output can fall under three cases: (i) the sample belongs to a positive query result (e.g., classifier decision = $x$ is a heavy hitter); (ii) the sample corresponds to a negative query result (e.g., decision = $x$ is not a heavy hitter); (iii) the decision on the query result cannot be taken with a high level of *confidence* (possible error). Only in case (iii) the portions of measurement data to which the sample pertains are directed to standard *aggregate-evaluate*, while in case (i) the predicted result is directly included in the query response.

**Workflow** As depicted in Fig 3, our workflow includes three phases: *training* phase, where the classification functions are built, *validation* phase, where classifiers are configured to preserve query result accuracy, and *run-time classification* phase, where classifiers are run in the wild to take decisions on incoming traffic. These are detailed below.

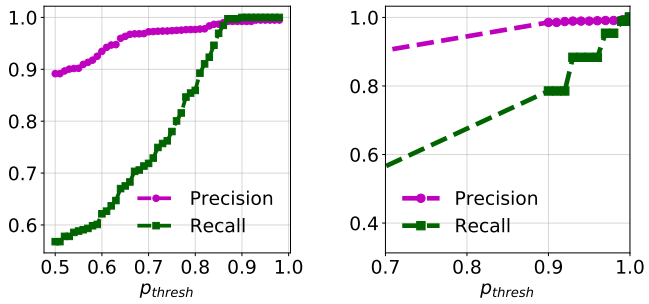
### B. Training

The training phase is divided in two steps: (i) *sampling*, which extracts training information (training sets) from recent measurement data, and (ii) construction of classification functions based on the obtained training sets.

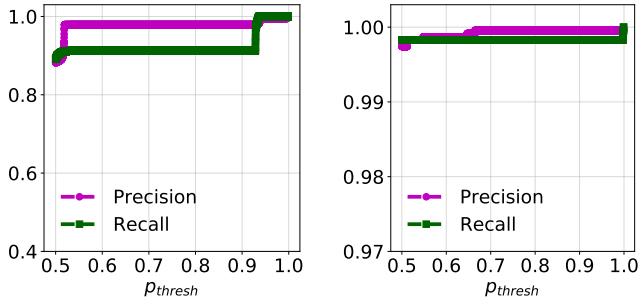
**Sampling** Training is performed using recent sets of raw measurements, *i.e.,* from previous query reporting intervals, where all query results are computed through standard *aggregate-evaluate*. In particular, the training set is created by sampling recent sets of raw measurements and by associating each sample with its ground-truth label represented as a binary indicator, e.g., the sample corresponds to a heavy hitter (1) or not (0)?

To make classifiers more efficient, sampling should ideally be a query-dependent operation. This is because the size of the samples and the way they are extracted (e.g., transformation of the raw measurement set) depend on specific query characteristics, such as its execution workflow or how selective it is. In this work, we propose two sampling methods that cover a wide range of queries.

The first method is used when *aggregate-evaluate* contain a *cardinality*-based predicate, e.g., for *DDoS* in Table I. In this case, a random subset of the raw measurement sets is




(a) Heavy hitter detection (b) DDoS detection




(c) Slowloris detection (d) Bursty aggregate detection

Figure 4: Sensitivity analysis performed in *validation*

selected based on sampling factor $k$. The extracted values are then grouped on the *aggregate* key, e.g., the DstIP in DDoS detection, and each sample in the training set is a tuple containing the cardinality of a group and the associated ground-truth label. The second method covers cases where query predicates include aggregation functions such as *mean*, *sum*, *stddev* e.g., for *HH* in Table I. In this case, the initial set is first grouped on the *aggregate* key, e.g., the srcIP for *HH*. For each group, $K$ values are then randomly selected.

**Classification functions** Once formed, the training set is used to build the classification functions. These are modelled following the *sigmoid* function of logistic regression. Such a model is selected due to its high interpretability, independence from complex tuning, and low computational cost. For each input sample they return: (i) the predicted label $l_{predicted}$ (1 if the sample participates to a query-related event, 0 otherwise), and (ii) the probability estimates $p_0, p_1$ associated with each label (with $p_0+p_1 = 1$). Values of $p_1$ ($p_0$ respectively) indicate how likely a sample is to be labelled as 1 (0 respectively) in the ground-truth, and are used to quantify the *confidence* for individual classification decisions.

**Cost of classification** To quantify the cost of classifiers, we measure the CPU overhead of individual classification executions. Each of these consists in the update of the sigmoid function $\frac{1}{1+e^{-z}}$, where $z$ is a linear regression hypothesis of coefficients $\theta_0, \theta_1, ..., \theta_K$ and $K$ is the sample size. The results are shown in Fig. 5 in terms CPU cycles[1] for different values of the sample size $K$. As depicted in the figure, the cost is $O(K)$ for $K \geq 10$. For $K \leq 5$, the cost is driven

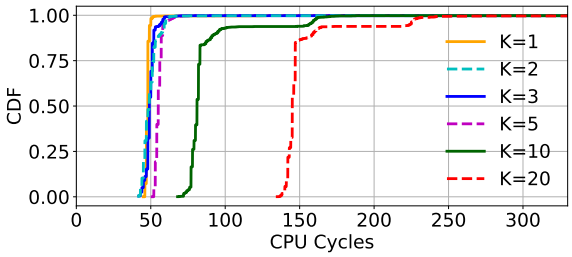



Figure 5: CPU overhead (cycles) of individual classifications for different sample sizes K.

by the computation of the $exp$ function (only significant operation). Note also that 50 cycles ($K \leq 5$), corresponding to approximately $17ns$ on the server used, is no more than the 25% of the time consumed for a single packet at 14.8Mpps (10 Gbps small pkts). However, classifications do not operate at the packet granularity, but at the one of query aggregation keys. As such, in the worst-case, *i.e.,* query result at the 5-tuple flow granularity and small reporting interval of 10ms, the classifier overhead is $\frac{17ns \cdot N_{flows}}{10ms}$, close to 0.1% of CPU time for 1000 active flows in the 10ms intervals [19]. Furthermore, overhead reductions (up to 45%) can be achieved by pre-computing values of the sigmoid function $\frac{1}{1+e^{-z}}$, thus reducing the execution of the classifier to the update of $z$ plus a small array lookup.

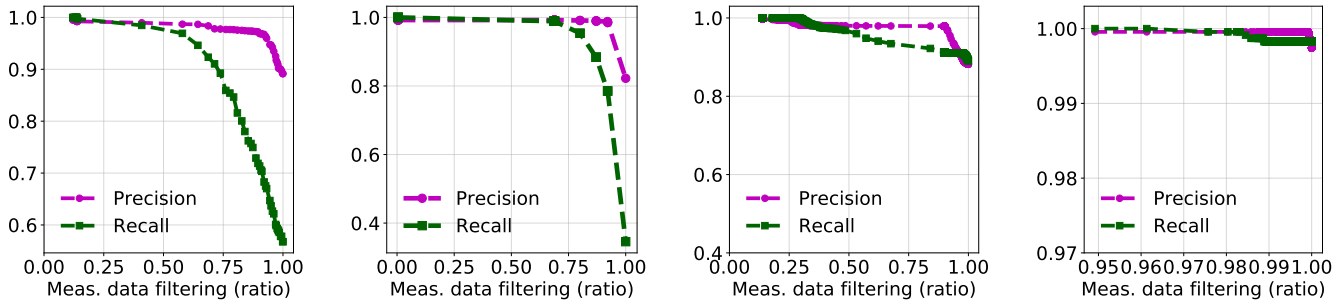
### C. Validation

The goal of the *validation* phase is to configure the constructed classifiers so that accuracy requirements can be met on the query results. Since classification functions are not error-free, accepting all $l_{predicted}$ labels in output may result in query result errors. This happens especially when the confidence on the classification is low, e.g., $(p_0, p_1) = (0.55, 0.45)$.

To configure the classifiers, we need to determine the right ranges of probability estimates $p_0, p_1$ under which the predicted labels $l_{predicted}$ in output from the classification functions can be "safely" accepted. More specifically, to express configurations, we use a probability threshold $p_{thresh}$, such that the classification result is accepted when $max(p_0, p_1) \geq p_{thresh}$. $p_{thresh} = 0.5$ corresponds to the baseline (all $l_{predicted}$ labels accepted). Intuitively, this setup maximizes the filtering, *i.e.,* the amount of raw data excluded from *aggregate-evaluate*, but at the same time it also maximizes the risk of introducing errors in the query results.

The objective of the validation phase consists in finding appropriate $p_{thresh}$ operating coordinates. This is achieved by conducting a small-scale sensitivity analysis for each classification function, using additional sets of labelled samples (*validation* sets) whose total volume is below 25% of the training set [3]. To quantify the accuracy of query results we use the *Precision*, *i.e.,* the ratio of the monitored query-related events that are *true*, and the *Recall*, *i.e.,* the fraction of detected

[3]The choice reflects standard machine learning practices, where training/validation sets are 80/20 splits of total labelled-samples set




(a) Heavy hitter detection (b) DDoS detection (c) Slowloris detection (d) Bursty aggregate detection

Figure 6: Filtered raw measurement data

*true* events. Fig 4 shows examples of the analysis for the queries in Table I. As observed, the value of $p_{thresh}$ can have a significant impact on the query result accuracy, especially on the Recall (many events missed by the queries). In addition, different queries follow different trends, which indicates that query-specific $p_{thresh}$ settings are required.

To select $p_{thresh}$, we adopt a method inspired by *Receiver Operator Curve* (ROC) analysis [17], [8]. While traditional ROC analysis can involve costly tuning of the $p_{thresh}$ cut-off(s) based on a optimal criterion, our method operates by simply testing for each classifier a random set of $p_{thresh}$ values in $[0.5, 1.0]$. After each test, Precision and Recall from the sensitivity curves are compared against the query Precision/Recall requirements. The selected $p_{thresh}$ value is the lowest for which both Precision and Recall satisfy the thresholds.

### D. Run-time classification

Trained and configured classifiers are applied in the wild to pre-process the raw measurement data extracted for each active query. To extract samples of the initial raw measurements set, the same mechanism as the one used in the training phase is adopted. Each sample, corresponding to a specific *aggregate key* (e.g., a specific srcIP for HH detection), is processed by the classification function of the query, which returns a pair $[l_{predicted}, (p_0, p_1)]$. If $max(p_0, p_1) \geq p_{thresh}$, the result is accepted, *i.e.,* the predicted query result $l_{predicted}$ is accepted and the portion of raw data matching the sample *aggregate key* is discarded. Otherwise, the measurement data is redirected to the *aggregate-evaluate* functions for standard processing.

## IV. Evaluation

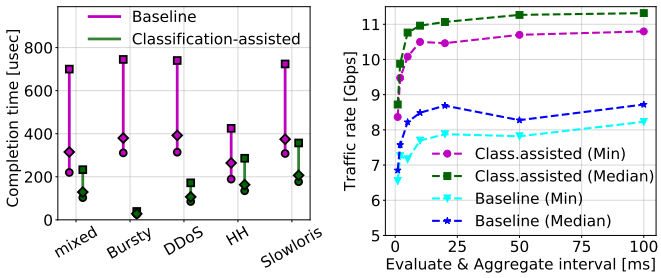
To investigate the benefits of classification-assisted query processing, we integrate our approach to a traffic monitoring implementation based on software packet-processing, which is running on a single CPU core[1]. The tool is based on the framework in [16][1] and relies on a hash table, indexed on the flow 5-tuples, to buffer raw measurements extracted from the traffic stream. Query responses are generated in short reporting intervals $T$ (default value, $T = 20ms$). At run time, measurement data samples (obtained from sampled flow-entries) are applied to the trained classifiers. If the classification decision is accepted, the flow-entries matching the *aggregate key* of the sample are excluded from further processing, and the predicted label is kept for the query response. For the remaining flow-entries, measurements are instead grouped by *aggregate key* and checked against the query predicates, which corresponds to the baseline *aggregate-evaluate* workflow. The classification functions are trained using samples from a 10s CAIDA traffic trace [2], reserving 2s for *validation*. For the four queries in Table I, the $p_{thresh}$ value obtained from the validation is, respectively, $0.86, 0.97, 0.92$ and $0.99$. The classifiers are then used to process 30 minutes of CAIDA traffic without additional training. The sampling setup is fixed for the duration of the experiments with $k = 10\%$ and $K = 5$.

### A. Measurement data volume

We first assess the performance of our approach in terms of the data volume involved in query processing. To this end, we run experiments with different $p_{thresh}$ setups for each classifier, measuring the effect of *filtered* data on *Precision* and *Recall*. The results are depicted in Fig. 6 and show that with an accuracy above $98\%$ (Precision and Recall), $50, 55, 70$ and $98\%$ of measurement data can be filtered for each of the four queries, respectively. The level of filtering achieved is significant, despite using a small 10s training set, but different query behaviors can be observed. These reflect a variety of query and traffic features (e.g., traffic distribution characteristics such as skewness and entropy) that make it more or less difficult to discriminate between positive and negative samples. Interestingly, the amount of data that can be safely filtered does not reflect the query *selectivity* level discussed in Sect. II-B. While *Slowloris* is generally the most selective query in Fig. 2, no more than 50% of its measurement data can be filtered as confidence on classifications is limited.

### B. Query processing cost

Since our implementation is based on software packet processing, we measure the processing cost in terms of CPU time ($t_{proc}$) consumed to craft query responses from the raw data per reporting interval. To experiment with different query workloads, we split the flow-address space and assign one query to each /12 prefix. In a *mixed* workload the query is randomly selected from Table I, while in other workloads




(a) $t_{proc.}$ (min, mean, max) (b) Supported traffic rate

Figure 7: Benefits on the monitoring pipeline used

considered all queries are of same type. For all experiments performed, the classifiers are configured to achieve an accuracy of 98% for both Precision and Recall.

As shown in Fig. 7a, nearly 60% of CPU time is saved on average for a *mixed* workload, while the gain obtained for specific query types is in accordance with the filtering ratios in Fig. 6, e.g., the processing time for *Bursty* is reduced by more than 10x on average, as more than 90% of measurement data is filtered by classifiers. Reduced processing time can translate to more query responses handled over a reporting interval. In particular, assuming constant time for raw data extraction[4], the guaranteed (minimum) number of simultaneous queries supported by our implementation is more than 3x higher than the baseline in *mixed* workload conditions.

Reduced query processing times can also improve the traffic speed supported by monitoring. To evaluate this, we split the experiment in 1 minute chunks, and for each one we measure the maximum traffic rate handled by the monitoring implementation (without dropping packets). As Fig. 7b shows, the classification-assisted approach can significantly speed up the monitoring pipeline, with gains in traffic rate up to 30%. Such gains (e.g., +3 Gbps) are higher than the ones obtained (see Sec.II-B) by fine-tuning the reporting interval ($\leq +1.5$ Gbps).

### C. Monitoring accuracy

Finally, we evaluate how effective our solution is in meeting the requirements of query result accuracy. We select a target accuracy of 98% (Precision and Recall) to be used in the validation phase, and for each 1-minute chunk we compute the deviation from this threshold. For example, a $+0.01$ Recall deviation corresponds to 99% Recall. As shown in Fig. 8, our approach satisfies, on average, the desired accuracy levels for all queries. Negative deviations, if any, never exceed $-0.02$. Interestingly, this result is based on a small $10s$ training set.

## V. Conclusion

We have proposed an approach to reduce the cost of monitoring query processing based on intelligent filtering of measurement data using lightweight classifiers. Through representative query examples and a real traffic trace, we

[4]In our implementation, this time is dominated by hashing and flow-entry retrieval executed for each packet, irrespectively of the query workload

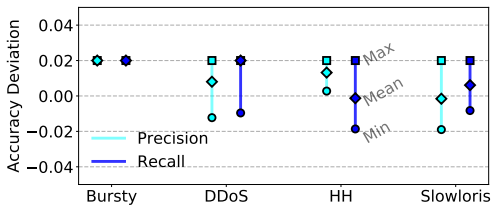



Figure 8: Accuracy deviations from 98% target(s)

demonstrated that our solution can effectively reduce processing workloads while preserving reporting accuracy, using no more than 10s training for 30min traffic. Future work will explore dynamic approaches for setting the training duration and re-training frequency based on traffic properties (e.g., for different times of the day), and investigate the use of our approach on heterogeneous network telemetry architectures.

## References

[1] "Trumpet," 2016. [Online]. Available: https://github.com/USC-NSL/Trumpet
[2] "The caida ucsd anonymized internet traces dataset - march 2018," 2018. [Online]. Available: http://www.caida.org/data/passive/passive_dataset.xml
[3] O. Alipourfard *et al.*, "Re-evaluating measurement algorithms in software," in *Proceedings of HotNets-XIV*, 2015.
[4] B. Arzani *et al.*, "Taking the blame game out of data centers operations with netpoirot," in *Proceedings of SIGCOMM '16*, 2016.
[5] S. Chaudhuri *et al.*, "Approximate query processing: No silver bullet," in *Proceedings of SIGMOD '17*, 2017.
[6] G. Cormode *et al.*, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, 2005.
[7] G. Cormode. *et al.*, "Finding frequent items in data streams," *Proc. VLDB Endow.*, vol. 1, no. 2, Aug. 2008.
[8] C. Cortes and M. Mohri, "Auc optimization vs. error rate minimization," in *Proceedings of NIPS'03*, 2003.
[9] A. Gupta *et al.*, "Sonata: Query-driven streaming network telemetry," in *Proceedings of SIGCOMM '18*, 2018.
[10] Q. Huang. *et al.*, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of SIGCOMM '17*, 2017.
[11] Q. Huang *et al.*, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of SIGCOMM '18*, 2018.
[12] A. Khandelwal *et al.*, "Confluo: Distributed monitoring and diagnosis stack for high-speed networks," in *Proceedings of NSDI'19*, 2019.
[13] A. Metwally *et al.*, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of ICDT'05*, 2005.
[14] M. Moshef *et al.*, "Scream: Sketch resource allocation for software-defined measurement," in *Proceedings of CoNEXT '15*, 2015.
[15] M. Moshref *et al.*, "Dream: Dynamic resource allocation for software-defined measurement," in *Proceedings of SIGCOMM '14*, 2014.
[16] M. Moshref. *et al.*, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of SIGCOMM '16*, 2016.
[17] M. Mozer *et al.*, "Prodding the roc curve: Constrained optimization of classifier performance," in *Proceedings of NIPS'01*, 2001.
[18] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proceedings of SIGCOMM '17*, 2017.
[19] A. Roy *et al.*, "Inside the social network's (datacenter) network," in *Proceedings of SIGCOMM '15*, 2015.
[20] K. Xie *et al.*, "On-line anomaly detection with high accuracy," *IEEE/ACM Transactions on Networking*, vol. 26, no. 3, 2018.
[21] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proceedings of NSDI'13*, ser. nsdi'13, 2013.
[22] L. Yuan *et al.*, "Progme: Towards programmable network measurement," *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, Feb. 2011.
[23] Y. Yuan *et al.*, "Quantitative network monitoring with netqre," in *Proceedings of SIGCOMM '17*, 2017.