ProgME: Towards Programmable Network MEasurement

Lihua Yuan Chen-Nee Chuah Prasant Mohapatra
University of California, Davis

***Abstract*—[1]Traffic measurements provide critical input for a wide range of network management applications, including traffic engineering, accounting, and security analysis. Existing measurement tools collect traffic statistics based on some pre-determined, inflexible concept of "flows". They do not have sufficient built-in intelligence to understand the application requirements or adapt to the traffic conditions. Consequently, they have limited scalability with respect to the number of flows and the heterogeneity of monitoring applications.**

We present ProgME, a *Prog*rammable *ME*asurement architecture based on a novel concept of *flowset* – arbitrary set of flows defined according to application requirements and/or traffic conditions. Through a simple flowset composition language, ProgME can incorporate application requirements, adapt itself to circumvent the scalability challenges posed by the large number of flows, and achieve a better application-perceived accuracy. The modular design of ProgME enables it to exploit the surging popularity of multi-core processors to cope with 7 Gbps line rate. ProgME can analyze and adapt to traffic statistics in real-time. Using sequential hypothesis test, ProgME can achieve fast and scalable heavy hitter identification.

I. Introduction

Accurate measurement of network traffic is a keystone of a wide range of network management tasks, e.g., traffic engineering, accounting, network monitoring, and anomaly detection. A measurement tool, be it a dedicated hardware or software running on routers or firewalls, collects statistics of network traffic. Management applications use these statistics to make network control decisions, such as rerouting traffic, charging customers, or raising alarms to administrators. The insights gained from traffic measurement are invaluable to administrators in making informed decisions on network planning or operations.

Fundamentally, traffic measurement involves counting the number of packets (or bytes) that satisfy some criteria over a particular period of time. Figure 1 provides a high-level conceptual diagram of traditional measurement architectures that typically try to find a matching flow for every sampled packet and increases the corresponding counter. A flow can be defined by 5-tuples in the IP headers. Example systems that adopt this approach include NeTraMet [2], FlowScan [3], and sFlow [4]. Such per-flow traffic statistics might be, upon a triggering event like the expiration of a timer or passing of a threshold, delivered to

[1]An earlier version of this paper appeared on [1].

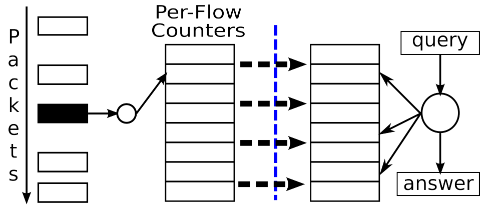


Figure 1: Traditional Measurement Architecture.

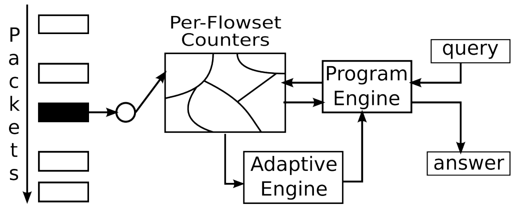


Figure 2: Programmable Measurement Architecture.

a centralized storage location. A management application, be it a network manager or an anomaly detection tool, can then perform post-processing on the per-flow statistics to retrieve useful information. For example, to answer user queries like "How much traffic goes to a particular network?", one can perform *selective aggregation* to count all the flows belonging to this query. Monitoring applications, e.g.,, heavy hitter identification, can search through the per-flow traffic statistics to find the elephant flows[2].

Although the traditional measurement architecture has had some success in offering insights about network traffic, its scalability is limited in practice for various reasons. First, the traditional architecture collects statistics based on an inflexible definition of flow. In today's high-speed networks, especially the core networks, the number of flows can easily reach millions. Keeping per-flow traffic profiles is challenging for memory and processor [5]. Even if one assumes that per-flow traffic profiles could be managed locally, delivering it to a remote server and storing it over a prolonged period of time incurs significant network and storage overhead. Second, the traditional architecture requires a post-processing approach. Measurement tools have little knowledge about the actual requirement of the management applications and focus only on providing sufficient statistics. It is up to the management applications to process per-flow traffic statistics and extract meaningful information. This disconnection between measurement

[2]Elephant flows are defined as the largest n flows in terms of number of packets or bytes generated.

tools and management applications forces the measurement tools to collect all the statistics that *might* be useful and at the finest granularity to meet the requirement on accuracy. Third, the traditional architecture cannot adapt itself to changing network conditions. It is difficult for a measurement tool engineered to monitor a few large flows (elephants) to re-adapt itself to monitor large number of small flows (mice), e.g., in the case of Distributed Denial-of-Service attack.

This paper calls for a departure from the traditional architecture using per-flow traffic profiles and the post-processing approach. We present a *Prog*rammable *ME*asurement architecture (*ProgME*) that can adapt to application requirements and traffic conditions in real time. Figure 2 shows the major components of ProgME. Our first proposal is to use a versatile definition of *flowset* – arbitrary set of flows – as the base of traffic statistics collection. In other words, ProgME keeps one counter per flowset. Compared to per-flow traffic statistics, per-flowset statistics enables one to achieve multiple resolutions within a traffic profile. Since flowsets can be defined arbitrarily, they do not necessarily map to the same number of unique flows or traffic volume. Therefore, one can track higher resolution statistics to maintain the desired accuracy for a sub-population of network traffic, while collecting coarse-grained aggregate statistics for the remaining traffic (e.g., through a flowset that catches uninteresting traffic) to reduce total number of counters required. Furthermore, since a flowset can contain arbitrary set of flows, one can construct flowsets that directly reflect the interest of management applications. For example, one can use a single counter to track packets from an invalid source IP address instead of keeping a large number of per-flow counters and aggregating them later.

The second key component of ProgME is a program engine that can dynamically (re)-program the definitions of flowsets based on user queries. By enabling the management applications to program the measurement tool, one can *pre-process* application requirements so that the tool only collects statistics that are directly useful to applications and at a desired granularity, thus significantly improving its scalability and performance. Note that we do not claim that collecting statistics according to user requirement is the right solution for all measurement tasks. ProgME can be most beneficial if users know their requirements beforehand. However, if one fails to envision the usefulness of certain traffic metrics and does not measure them directly in the first place, a posteriori analysis on aggregate data might not generate accurate estimates of these metrics.

ProgME is intended as an *on-line* measurement module and offers the flexibility to support adaptive measurement algorithms. To match the requirement of high-speed links, ProgME features a modular system design that can effectively exploit the surging availability of multi-core processors to reach approximately 7 Gbps without sampling. Its program engine can merge or partition flowsets and re-allocate the counters dynamically based on past traffic statistics to increase tracking accuracy and measurement efficiency. We believe ProgME can be deployed on both dedicated measurement devices and backbone routers – although the latter would require some adaption of the algorithm at the hardware level. This paper focus on ProgME as a single box solution, therefore distributed traffic measurement and analysis is beyond the scope of this paper.

The contributions of this paper are as follows:

- We propose a versatile *flowset* definition as the base unit of network measurement. We present a Flowset Composition Language (FCL) for defining flowsets consisting arbitrary set of flows and a binary decision diagram (BDD)-based data structure for efficient set operations and packet matching (Section II).
- We show that the flexibility offered by our flowset definition is helpful in broad categories of network measurement, including answering user queries (Section III) and identifying heavy hitters (Section IV).
- We propose a scalable Flowset-based Query Answering Engine (FQAE) in (Section III) to support arbitrary user queries. Used in conjunction with sampling, FQAE can achieve the same accuracy for any given set of queries compared to an ideal flow-based measurement approach, while achieving orders of magnitude cost reduction in terms of memory requirements.
- We propose a Multi-Resolution Tiling (MRT) algorithm, which dynamically re-program the flowset measurement to zoom in on heavy hitters (Section IV). It can identify heavy hitters under tight memory budget by re-defining *flowsets* and re-allocating the associated counters. MRT analyzes the traffic and the statistics collected sequentially and can be deployed on-line.

We describe our design rationales and the three major components of ProgME: Flowset Composition Language (FCL) in Section II, Flowset-based Query Answering Engine (FQAE) in Section III, and Multi-Resolution Tiling (MRT) algorithm in Section IV, respectively. Section V presents the performance evaluation results of the ProgME framework. We discuss related work in Section VI and conclude the paper in Section VII.

II. Arbitrary Flowset

Table I summarizes the notations used in this paper.

Symbol	Explanation
f	A flow
H	Set of fields that defines "flow"
F	A flowset
F_w	The weight of a flowset F
F_c	The counter associated with F
P	A packet enumerator, either a trace file or live traffic
Q	A list of user queries
$\mathbb{U}$	The universal set of flows

Table I: Notations.

Traditionally, network statistics are collected based on the concept of *flows*. A *flow* f refers to a set of packets that have the same n-tuple value in their header fields. Let $H : \{H_1, H_2, \cdots, H_n\}$ denote the header fields used in the flow definition. Typical definitions of flow include the 5-tuple of $H : \{prt, sip, spt, dip, dpt\}$ or the 2-tuple of $H : \{sip, dip\}$ in which prt is the protocol field, sip and dip are the source and destination IP address and spt and dpt are the source and destination port, respectively. Other header fields, e.g., Type-of-Service (TOS), could be used as well. A flow is often used as the base unit for traffic measurement. With a n-tuple definition, a flow can be regarded as a point in the n-dimension space with each field as a dimension.

In the context of packet classification (including routing and packet filtering), it is often necessary to designate an action (e.g., route to a certain interface, filtering the packet) to a *set* of flows. The status quo is the concept of *superflow*, which takes a similar form of the definition of flow except each field is extended to a *range* of values. In the general 5-tuple superflow $H' : \{prt_r, sip_r, spt_r, dip_r, dpt_r\}$ definition, sip_r and dip_r are CIDR address blocks and prt_r, spt_r, and dpt_r could be value intervals. The semantics of superflow is not flexible enough — it is restricted by the well-defined structure and can only describe a *regular-cut* set of flows, where each field contains a contiguous range of values. Therefore, sip_r and dip_r should contain IP addresses that form a valid CIDR block with contiguous IP addresses, while spt_r and dpt_r should contain continuous interval of integer values. For example, if one is to visualize a 2-tuple superflow defined by sip_r, spt_r on a 2-dimension space, superflow can only carve out rectangles of various size [6], as shown by the solid and dotted boxes in Figure 3.

We define a flowset to be a set of *arbitrary* flows. A flowset is not limited by the structure of superflow and can take any shape, even being segmented in the space (as one shall see shortly). A flow can be considered a special case of flowset containing only one member. To the best of our knowledge, there are no existing languages for specifying such a versatile flowset other than an inefficient enumeration of superflows.

In the following part of this section, we first present a flowset composition language (FCL), which enables user to specify an arbitrary set of flows as a single entity (Section II-A). Section II-B clarifies related definitions and Section II-C introduce a canonical representation of flowset using binary decision diagram (BDD). Coupled with the underlying BDD representation of flowsets, FCL allows users to specify their requirement on aggregated traffic statistics and enables measurement tools to preprocess user requirements.

A. Flowset Composition Language (FCL)

We present a simple Flowset Composition Language (FCL) using set algebra to enable specification of arbitrary flowset (Table II). The primitive of FCL (pr) is the 5-tuple superflow definition, which by itself is a flowset that defines a *regular-cut* set of flows. One can use other primitives

e	$=$	$e \text{ op } e \mid (e) \mid \neg e \mid pr$
op	$::=$	$\cap \mid \cup \mid \backslash$
pr	$::=$	$< prt, sip, spt, dip, dpt >$

Table II: Grammar of Flowset Composition Language.

F_1: Traffic from private IP
$F_1 = r_1 \cup r_2 \cup r_3$, where
$r_1 =< *, 10./8, *, *, * >$
$r_2 =< *, 172.16./12, *, *, * >$
$r_3 =< *, 192.168./16, *, *, * >$
F_2: FTP not from 10.1./16
$F_2 = (x_1 \cup x_2) \cap \neg x_3$, where
$x_1 =< *, *, *, *, 20 >$
$x_2 =< *, *, *, *, 21 >$
$x_3 =< *, 10.1./16, *, *, * >$

Table III: Sample Flowsets.

as long as it specifies a set of flows. We choose the 5-tuple definition because of its wide usage in the context of firewall configuration and policy-based routing.

The FCL grammar defines several standard binary set operators (op), e.g.,, *intersection* ($\bigcap$), *union* ($\bigcup$), and *relative complement* ($\backslash$), and unary operators like *a*bsolute complement ($\neg$). The expression of a flowset (e) can just be a primitive itself (pr). One can apply one of binary operators to two flowsets (e op e) and assign the result to another flowset. One can also apply one of the unary operators to one flowset and assign the result to another flowset. Note that this grammar is recursive and one can use the parentheses operator, (), to change the precedence. These operations are sufficient to build a flowset with arbitrary set of flows. In addition, one can, using the operators provided, build more complicated logical operations, e.g., NAND or NOR. All the laws associated with set algebra, including the *commutative*, *associative*, *distributive*, *identity*, and *complement* laws, apply to flowset as well.

Table III presents two examples of such flowsets that might be of practical interest to the administrators. Flowset F_1 presents all flows originated from private address space. In practice, administrators are interested in tracking these flows because packets with unroutable IP address are not legitimate and are often used by attackers and spammers. Flowset F_2 presents incoming FTP traffic (port 21/22) except those from an internal network. F_1 and F_2 are depicted in Figure 3 in dashed- and solid-line rectangles, respectively. Notice that a flowset, as a single entity, can cover disconnected and irregular parts in the universal set.

B. Definitions

Since flowset is a type of *set*, concepts and definitions in set theory apply here. In the following, we highlight the definitions that are useful for our discussions.

- The universal flowset $\mathbb{U}$ contains all the possible flows, and the empty flowset $\emptyset$ contains no flow. Two flowsets A and B are said to be *disjoint* if their intersection is empty, i.e., $A \bigcap B = \emptyset$.
- We denote the cardinality of flowset F as $|F|$, which is a measure of the "number of possible flows of the

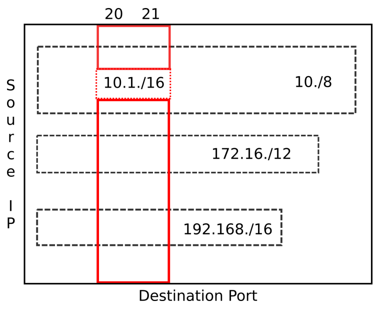


Figure 3: Visualization of Table III.

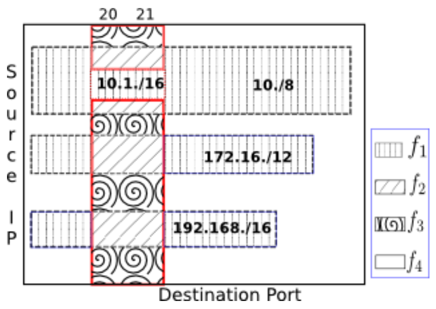


Figure 4: Disjoint Flowsets of Figure 3.

flowset". Note that $|F|$ can be larger than the actual number of active flows one observes in a particular traffic instance, which we denoted as $|F|'$.

- We denote $|H_i|$ as the total number of possible values of field H_i. Therefore, $|H_{sip}| = |H_{dip}| = 2^{32}$, $|H_{spt}| = |H_{dpt}| = 2^{16}$, $|H_{prt}| = 2^8$. The total number of possible flows, which is also the cardinality of $\mathbb{U}$, is $\prod_{i=1}^{n} |H_i|$.
- A set of flowsets $\mathcal{F} : \{F_1, F_2, \cdots, F_n\}$ is said to be a *partition* of a flowset X *iff* (Eq. 1) none of the flowset in $\mathcal{F}$ is empty, (Eq. 2) flowsets in $\mathcal{F}$ are pair-wise disjoint, and (Eq. 3) the union of all flowsets in $\mathcal{F}$ equals to X. In particular, $\mathcal{F}$ is *complete* if it is a partition of $\mathbb{U}$.

$$F_i \neq \emptyset \qquad \forall\, 1 \leq i \leq n \tag{1}$$
$$F_i \bigcap F_j = \emptyset \qquad \forall\, 1 \leq i \neq j \leq n \tag{2}$$
$$\bigcup_{i=1}^{n} F_i = X \tag{3}$$

- We denote F_c as the counter associated with a flowset F. The counter is updated when a matching packet is observed and can take any unit, e.g., packets or bytes. We also denote F_w the actual weight of F, and F'_w the measured weight of a flowset F. Measuring F_c and F'_w are equivalent if one keeps a counter for all packets.

C. Underlying Data Structure

The string representation of flowset is not ideal for complicated set operations. Following the approach used to encode firewall rules and access lists in recent studies [7, 8], we use binary decision diagram (BDD) [9] as the underlying data structure for flowset (referred to as *flowset label* hereafter). BDD is an efficient data structure that is widely used in formal verification and simplification of digital circuits. A BDD is a directed acyclic graph that can compactly and canonically represent a set of boolean expressions. In a BDD graph, the non-terminal vertices represent the variables of the boolean function, and the two terminal vertices represent the boolean values 0 (True) and 1 (False).

In ProgME, we map every bit field of packet header to a BDD variable. For example, we encode the source IP block 128.0.0.0/4 as $sip(s_1 s'_2 s'_3 s'_4)$, whose corresponding BDD is shown in Figure 5a. Similarly, the BDD for source

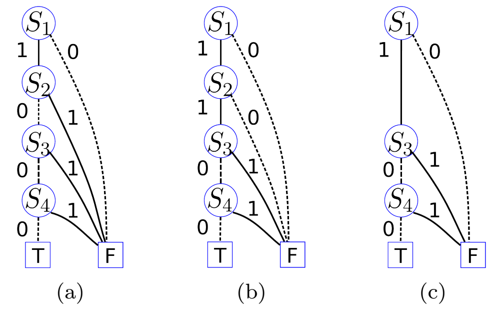


Figure 5: BDD Encoding.

IP 192.0.0.0/4 is depicted in Figure 5b. Note that only the first 4 bits are used and the 24 masked bits are omitted in the BDD. Performing set operations such as *intersection*, *union*, *not*, and *implication* using BDDs is straightforward. Figure 5c depicts the union of Figure 5a and Figure 5b. To determine if a packet matches a flowset, one can extract corresponding bit value from packet header and traverse through the BDD once until reaching either the "true" or "false" node. For example, a packet with source IP "200.0.0.1" can be quickly identified to be not in the flowset depicted in Figure 5c since traversing with $s_1 = 1$ and $s_3 = 1$ leads to false.

The BDD representation of flowset has the following properties.

- The number of BDD variables used, V, is a constant defined by the size of the defining variables. For the 5-tuple superflow predicate, it is 104 (8 bits protocol, 2x32 bits source and destination IP address and 2x16 bits source and destination port).
- The number of BDD nodes used to describe a 5-tuple flow (N_f) is 104 as every bit variables is used.
- The number of BDD nodes used to describe a 5-tuple superflow (N_s) has an upper-bound of 104. This is because BDD ignores the unused bit variables, e.g., the masked bits in CIDR IP address block.
- Since a flowset is formed by set operation among a number of N superflows, the number of BDD nodes used to describe any flowset has an upper-bound that is determined by the total number nodes used to define each flow. The actual number of nodes can be smaller since BDD keeps the canonical form. The depth of the flowset has an upper-bound equavilent to the longest of the defining superflow.

III. Flowset-based Query Answering Engine (FQAE)

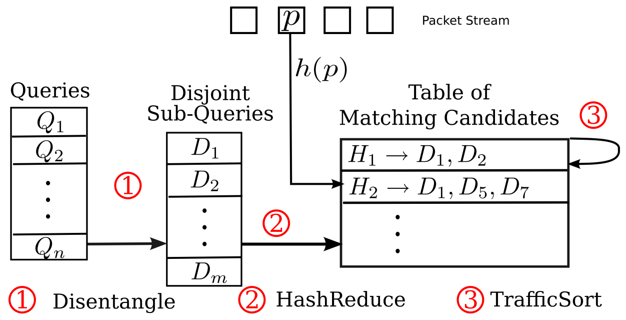


Figure 6: Flowset-based Query Answering Engine.

A major task of traffic measurement is to answer user queries about the characteristics of certain traffic aggregates. These traffic aggregates can have very different granularities. For example, one might query for the FTP traffic to certain hosts (fine granularity) or a particular ingress-egress pair of the network (coarse granularity). As discussed in Section I, current measurement systems collect fine-grained per-flow traffic statistics and rely on individual applications to perform *post-processing* to extract the desired information. This approach is not scalable since modern networks could observe millions of flows.

We make the observation that the total number of potential user queries can be far smaller than the number of flows it observes. If the measurement system has sufficient knowledge about the queries, it only needs to maintain *aggregated* state information that pertains to the queries, thereby avoiding the expensive per-flow states. In the following section, we present the Flowset-based Query Answering Engine (FQAE) that is capable of answering any user queries on traffic aggregates while maintaining a minimum number of counters. FQAE contains two fundamental blocks – a measurement engine that collects per-*flowset* traffic statistics and a program engine that takes a list of user queries (Q) as input and controls what to measure. The user queries are written using FCL as illustrated in Table III.

To collect per-flowset traffic statistics, one needs to increase the counter associated with a flowset upon observing a matching packet. This is similar to the classic packet classification problem, but has the following distinct differences. In packet classification, the goal is to find the *best* matching rule. Multiple rules can match a given packet, but a conflict resolution mechanism, e.g., longest-match-first in routing or first-match-first in packet filtering, can be used to determine the best matching rule. Once the best-matching rule is found, other rules can be safely ignored. In our case, one packet might need to be counted for multiple matching flowsets since queries might have non-empty intersections.

One naive approach is to match a packet against all queries one-by-one. This is inefficient when the number of queries is large. As illustrated in Figure 6, our approach is to first disentangle the user queries to disjoint sub-queries such that each packet matches exactly one sub-queries (details in Section III-A). Consequently, we only need to find the only matching sub-query and increment its counter. We follow the similar approach as in EGT-PC [10] to achieve this goal (details in Section III-B). One can certainly imagine using other packet classification techniques to assist in identifying the matching sub-query. However, the tradeoffs in memory and speed need to carefully explored. Please refer to [11] for a comparison of some packet classification algorithms.

Algorithm 1: $D \leftarrow$ Disentangle(Q).

```
input  : A list of queries $Q$ ($|Q| = n > 0$)
output: A list of disjoint flowsets $D$
1  $D.append(\mathbb{U})$;
2  foreach $x$ in $Q$ do
3      for $p$ in $D$ do
4          if $x <> p$ then                          // identical
5              break ;
6          else if $x \subset p$ then                 // subset
7              $D.append(p \setminus x)$ ;
8              $D.replace(p, \quad x)$;
9              break ;
10         else if $x \supset p$ then                 // superset
11             $x \leftarrow x \setminus p$ ;
12         else if $p \bigcap x \neq \emptyset$ then  // overlap
13             $D.append(p \setminus x)$ ;
14             $D.replace(p, \quad x \bigcap p)$ ;
15             $x \leftarrow x \setminus p$ ;
16         else continue ;                            // disjoint
17     $D.append(x)$ ;
```

A. Disentangling User Queries

Algorithm 1 generates disjoint sub-queries (D) from a list of user queries (Q). It works by adding the flowsets in Q to D in sequence. For every flowset in Q, we compare it with flowsets in D in sequence. A pair of flowsets must satisfy one of the following relationship: identical (line 3), subset (line 6), superset (line 10), overlap (line 12), and disjoint (line 16). Therefore, one can use set operations to separate the overlapped flowsets. Note that Algorithm 1 initiates D with one flowset – the universe ($\mathbb{U}$). As user queries might not cover the universe, this step ensures the resulting D is complete (a partition of $\mathbb{U}$). Consequently, every packet will match exactly one flowset in D.

Figure 4 illustrates the effect of running Algorithm 1 on the two queries in Table III (shown in Figure 3). The two flowsets defining the original queries have non-empty intersection. They divide the universe into four disjoint flowsets. Note that all operations in Algorithm 1 are performed using the underlying BDD-based data structure.

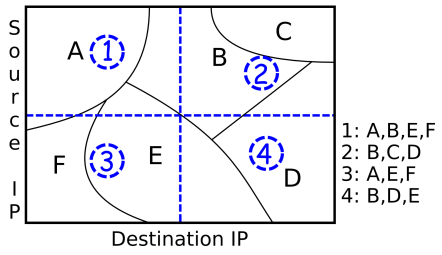


Figure 7: *HashReduce* Algorithm.

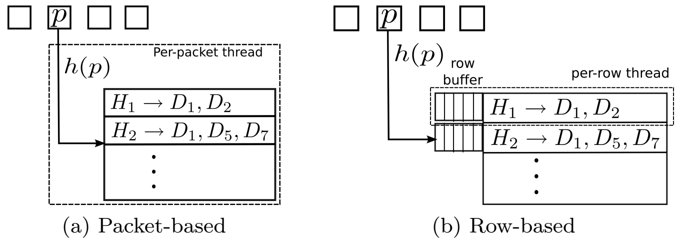


Figure 8: Parallel Versions of ProgME

B. Reducing Matching Candidates

Since the disjoint sub-queries D is a partition of the universe $\mathbb{U}$, every packet is guaranteed to match exactly one flowset. However, the naive approach — comparing a packet against each flowset until a match is found — is still not an efficient solution when the number of flowsets in D is large.

FQAE introduces a hash table based mechanism called *HashReduce* to reduce the number of comparisons required to find the matching flowset (step 2 in Figure 6). We use a hash function that simply extracts several bits from the header fields. For every possible hash value H, we build a BDD H_{bdd}, which describes a flowset containing all flows with this particular value in the header. The table of matching candidates can then be built by finding all flowsets in D that has non-empty intersection with H_{bdd}. Consider Figure 7 as an example that uses the first bit from source IP and the first bit of destination IP field. The hash function, by extracting two bits, divides the universe into four quadrants, each intersects with a few flowsets in D.

The *HashReduce* mechanism follows the similar spirit as EGT-PC [10], which uses one or two header fields to find candidate matching rules in n-tuple packet filter. Furthermore, it presents a tradeoff between memory and lookup speed that can be fully customized. Using more bits in the hash function incurs more memory overhead but can reduce the number of candidates in table entries.

C. Traffic-aware Optimization

During the measurement process, FQAE performs *TrafficSort* (step 3 in Figure 6), which sorts the table of matching candidates based the packet counters of matching candidates. Consequently, candidates matching more packets will appear earlier in the process, thereby reducing computation overhead. Note that this seemingly simple optimization is possible only because FQAE make flowsets fully disjoint. If flowsets have non-empty intersections, finding the optimal order is NP-complete, and one will have to resolve to heuristics, as some have attempted in the context of packet filtering [12, 13].

D. Collecting and Reporting Statistics

Collecting traffic statistics is a simple two-step process. Upon receiving a packet, FQAE first uses the same hash function to extract the bits from the packet header and looks up the table of matching candidates for a list of candidate flowsets. Then, FQAE compares the packet to the flowset sequentially until a matching flowset is found. Based on the statistics collected for each sub-queries, answering user queries requires a simple aggregation. Note that the fundamental difference here, as compared to per-flow statistics, is that sub-queries are generated according to user queries and we expect the number to be significantly smaller than the number of flows in traffic.

E. Implementation

The speed of FQAE, in terms of how many packets it can process within one second, is an important metric for evaluating the feasibility of deploying ProgME in high-speed networks. The interpretation of FCL and query disentanglement are performed when user queries arrive. FQAE needs to perform the following two operations at per-packet level in order to match the packet to its corresponding flowset: (1) HashReduce, a hash function to identify a list of matching candidates. This step basically extracts several relevant bits from the packet header. (2) If there are n matching candidates, check the packets sequentially against the candidates until a matching flowset is identified. This requires $n - 1$ matching operations in the worst case.

To achieve best performance, the actual implementation of FQAE must adapt to the underlying platform. In the following, we explore several implementation strategies of FQAE. Our first implementation (*mFQAE*) is a generic implementation that works on any linux-based PC platform. *mFQAE* is a monolithic program that runs on a single CPU.

The recent trend in CPU development is moving towards multi-core processors instead of increasing clock speed. However, recent work on traffic measurement are yet to exploit parallelism with these multi-core processors to increase the processing speed.

One naive approach for exploiting multi-processor is packet-level parallelism (*pFQAE*, as illustrated in Figure 8a), in which one instantiates a thread for every packet. This approach basically relies on the underlying operating systems to create a ProgME instance for each packet, thereby distributing the load over multiple processors. However, our measurement suggests that this approach results in a system with worse performance than that of the original single-thread *mFQAE*. The reason is that even though thread is lightweight, they still incur overhead when they are created. In our scenario, thread creation incurs more overhead than matching one packet in ProgME. Therefore, doing packet-level parallelism is not feasible.

Our second approach is row-level parallelism ($rFQAE$), which is based on the worker thread pool pattern. In this version (Figure 8b), the main thread running the HashReduce routine will hash packets into their respective row, where they are queued in a per-row buffer. For each row, a worker thread is instantiated to process packets in its buffer. When the number of independent sub-queries are large for each row, this row-level parallelism enables the main process to quickly distribute the computation load to other processors available to the system. We evaluated both $mFQAE$ and $rFQAE$ later in our experiments (Section V-D) and found $rFQAE$ can effectively exploit the multi-core processors to improve performance.

IV. Heavy Hitter Identification

Heavy hitters, or elephants, are the largest-n flows in terms of weight in network traffic. Alternatively, one can define heavy hitters as flows with a weight larger than a threshold θ. These two notions are equivalent if the threshold θ equals the weight of the nth largest flow. In this paper, we use the latter definition unless mentioned otherwise. We further assume the weight of a flow f, f_w, is defined relatively as a percentage of total traffic. Identification of heavy hitters are of particular interest to network management. For example, traffic engineering often focus on re-routing the few heavy hitters instead of worrying about the large number of mice [5].

Identifying heavy hitter is trivial if one maintains a counter for every single flow. However, this naive approach is not memory-efficient and does not scale to large number of flows. We propose Multi-Resolution Tiling (MRT), which exploits the versatility of FQAE and offers scalable heavy hitter identification. Our key idea is that one can, by observing a flowset, *infer* the characteristics of its subsets or objects (the flows). Therefore, one can selectively zoom into flowsets that might contain heavy hitters while ignoring others.

A. Multi-Resolution Tiling (MRT)

Algorithm 2 presents the multi-resolution tiling (MRT) algorithm for identifying elephants. MRT starts from a range R, which is provided through user specification as a flowset. This enables one to only identify elephants within a certain flowset, e.g., elephants that are TCP flows. If no user specified R is given, MRT starts its search set to $\mathbb{U}$.

At each iteration, MRT calls upon FQAE to match a list of S packets from P. For every flowset F in D, MRT performs sequential hypothesis test (Section IV-B) to determine if the weight of F (F_w) is larger than θ, the threshold that defines an elephant. MRT uses the following logical inference rule (Eq. 4) to determine if a flowset can be ruled out in considering its possibility of having any elephants. The rule states: if the weight of a flowset F, which is the sum of the weight of all flows in it, is smaller than θ, then none of the flow f in F can possibly be an elephant.

Algorithm 2: Multi-Resolution Tiling.

```
input : $P$: a packet enumerator
input : $R$: a flowset defines the search range
output: $Eleph$: A list of identified elephants
$eleph \leftarrow \{\}$;
$mice \leftarrow \{\}$ ;
$D \leftarrow$ Partition ($R$) ;
repeat
    FQAE ($D$, $P$, $S$) ;
    for $F$ in $D$ do
        if $F_w < \theta$ then                // no elephants
            $mice$.append ($F$);
        else if $|F| = 1$ then              // elephant
            $eleph$.append ($F$) ;
        else if $F_w >= \theta$ then
            $D$.replace ($F$, Partition ($F$)) ;
    $D$.replace ($mice$, union ($mice$));
until $ElephantsFound$;
```

$$F_w < \theta \;\;\Rightarrow\;\; f_w < \theta \;\;\forall\;\; f \in F \tag{4}$$

If it is impossible for a flowset to contain any elephant, we exclude all flows in F from further consideration. Otherwise, we partition F into multiple disjoint flowsets and start another iteration. The partition algorithm will be discussed in great detail in Section IV-C. In essence, MRT keeps on filtering out flowsets that cannot contain elephants while zooming in on those that might. This iteration terminates when all the elephants are identified. For identifying threshold-θ elephants, this happens when all the flowsets with a weight larger than θ contain only one flow. For identifying largest-n elephants, this happens when the largest n flowsets contain only one flow.

B. Sequential Hypothesis Testing

In Algorithm 2, it is crucial to determine quickly if the weight of a flowset is larger than θ ($F_w > \theta$). We propose to use sequential analysis, more specifically *sequential probability ratio test* (SPRT) proposed by Wald [14], to achieve this. SPRT has been used successfully by Jung et al. [15] for port scan detection. Instead of using a fixed sample size to determine the correctness of a hypothesis, sequential analysis allows one to determine dynamically whether further observation is required based on the current observation.

Let H_0 be the null hypothesis and H_1 be the single alternative. An ideal test procedure should satisfy user requirement on false positive rate (α) and false negative rate (β) while requiring the minimum number of observations. SPRT, for all practical purposes, can be regarded as such an optimum sequential test procedure.

Let us denote the result of ith observation as X_i and the result of a series of n observations as a vector $X :<$

$X_1, X_2, \cdots, X_n >$. SPRT hinges on finding $\Lambda(X)$ — the probability ratio that this entire observation is produced when H_1 is true as compared to the case when H_0 is true.

$$\Lambda(X) = \frac{\mathbf{P}\{X|H_1\}}{\mathbf{P}\{X|H_0\}} \tag{5}$$

As described in Eq. 6, we compare $\Lambda(X)$ against two positive number A and B, where A and B are determined by the user prescribed strength (α, β) ($A \leq \frac{1-\beta}{\alpha}$, $B \geq \frac{\beta}{1-\alpha}$, $A > B$). If $\Lambda(X)$ is greater than A (smaller than B), we consider that there are strong enough statistical evidence to *accept* (*reject*) the null hypothesis and the test terminates. Otherwise, we *continue* with more observations.

Intuitively, $\Lambda(X)$ is an indicator of the likelihood whether H_0 or H_1 is true.

$$decision = \begin{cases} \text{reject } H_0 \text{ (accept } H_1) & \text{if } \Lambda(X) > A \\ \text{accept } H_0 \text{ (reject } H_1) & \text{if } \Lambda(X) < B \\ \text{continue observation} & \text{Otherwise} \end{cases} \tag{6}$$

In order to determine if MRT should zoom into a particular flowset F, we need to determine if the weight of F is larger than θ. Therefore, our null hypothesis H_0 is $F_w < \theta$ and the alternative hypothesis H_1 is $F_w \geq \theta$ (Eq. 7). Since these two hypotheses are composite hypotheses, the actual hypothesis we used for testing is H_0' and H_1' in Eq. 8. Note that θ^- ($\theta^- < \theta$) is chosen such that false positive rate is smaller than or equal to α. Similarly, θ^+ ($\theta^+ > \theta$) is selected such that false negative rate is smaller than or equal to β:

$$H_0 : F_w < \theta \quad \text{and} \quad H_1 : F_w \geq \theta \tag{7}$$
$$H_0' : F_w = \theta^- \quad \text{and} \quad H_1' : F_w = \theta^+ \tag{8}$$

For the ith packet (p_i) observed, we use X_i to indicate if it is a member of F:

$$\begin{cases} X_i = 1 & \text{if } p_i \in F \\ X_i = 0 & \text{if } p_i \notin F \end{cases} \tag{9}$$

Therefore, X_i is a Bernoulli random variable with parameter F_w.

$$\begin{array}{ll} \mathbf{P}\{X_i = 1|H_1'\} = \theta^+ & \mathbf{P}\{X_i = 1|H_0'\} = \theta^- \\ \mathbf{P}\{X_i = 0|H_1'\} = 1 - \theta^+ & \mathbf{P}\{X_i = 0|H_0'\} = 1 - \theta^- \end{array} \tag{10}$$

With n packets, one observes a vector of random variable $X :< X_1, X_2, \cdots, X_n >$. If these n packets are randomly sampled, then X_i are all independently and identically distributed (i.i.d). Therefore, $\Lambda(X)$ can be found as the product of the probability ratio of every single observation (Eq. 11). Eq. 12 defines $\Lambda(X)$ in log space, which is easier for computation, especially if $\Lambda(X)$ is incrementally updated.

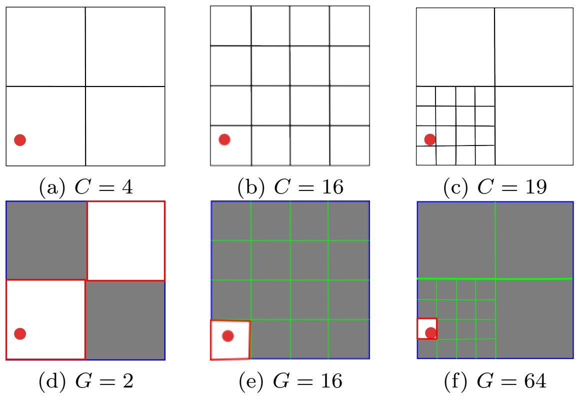


Figure 9: Partition Strategies.

$$\Lambda(X) = \prod_{i=1}^{n} \Lambda(X_i) = \prod_{i=1}^{n} \frac{\mathbf{P}\{X_i|H_1\}}{\mathbf{P}\{X_i|H_0\}} \tag{11}$$
$$\log \Lambda(X) = \sum_{i=1}^{n} \log \Lambda(X_i) \tag{12}$$

Let us denote the scenario that m among the n observed packets belongs to F as X_n^m. The probability of observing X_n^m when H_1 or H_0 is true can be found as:

$$\mathbf{P}\{X_n^m|H_1'\} = (\theta^+)^m(1-\theta^+)^{n-m} \tag{13}$$
$$\mathbf{P}\{X_n^m|H_0'\} = (\theta^-)^m(1-\theta^-)^{n-m} \tag{14}$$

One can determine the probability ratio of X_n^m as:

$$\log \Lambda(X_n^m) = m \log \frac{\theta^+}{\theta^-} + (n-m) log \frac{1-\theta^+}{1-\theta^-} \tag{15}$$

Eq. 15 requires the knowledge of n and m. Our *FQAE* routine counts the number of packets matched by each partition (m). n is simply the total number of packets observed so far. The value of $\log \Lambda(X)$ can then be compared with $\log A$ and $\log B$ as described in Eq. 6.

C. *Partition Strategies*

After determining that a flowset F might contain elephants, we need to partition F so that MRT can zoom into this flowset. The total number of possible partitions for a set F with n elements can be found recursively using Bell number with $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$ and $B_0 = B_1 = 1$. This number can be huge, even when the cardinality of F ($|F|$) is only marginally large, say 10. Therefore, it is impractical to explore every possible partition of F. Choosing a particular partition presents a tradeoff between memory consumption and speed in identifying elephants. We define the memory cost factor C as the number of subsets generated and the identification gain factor G as the cardinality of original flowset over the total cardinality of remaining flowsets that might contain elephants.

One natural strategy is to partition F into equal size subsets. Figure 9a and Figure 9b present two approaches with different memory cost factor and Figure 9d and Figure 9e present their respective results after one iteration.

	$\{sip\}$	$\{dip\}$	$\{sip,\ dip\}$
#1	53,191	214,411	336,463
#2	52,762	127,543	293,519

Table IV: Number of Flows in 5-minute Traces.

With a large memory cost factor, one can partition the flowsets into more smaller subsets. Consequently, it can exclude more flowsets in a single iteration and achieves a larger gain factor. Therefore, the optimal strategy is to use the largest memory cost factor as long as it satisfies the memory constraint. The number of iterations (N) required to identify the elephants is:

$$N = \log_C \ |\mathbb{U}| \tag{16}$$

Without a priori knowledge about the elephants, equal-size partition is the optimal strategy. In reality, however, administrators do have knowledge to make educated guesses, which might further improve the speed of heavy hitter identification. For example, one probably expect the protocol field of elephants to be TCP or UDP in most networks. For a particular network, certain IP addresses, e.g., Web/FTP server and certain port numbers e.g., port 21 or 80, are more likely to appear in elephants.

Using ProgME, it is easy to exploit user knowledge to improve identification of elephants. Our approach is to use preferential partitioning, which allow users to predefine flowsets with an amplified memory cost factor C'. This is illustrated in Figure 9c. The lower left quadrant is assigned a larger memory cost factor and is therefore partitioned into smaller subsets. Consequently, even though the memory consumption of the strategy in Figure 9c is lower than the strategy in Figure 9b, the identification gain is larger.

The effectiveness of preferential partitioning relies heavily on the users making correct guesses. We believe this is a reasonable assumption for administrators that monitor network behavior on a daily basis.

V. Evaluation & Discussion

In this section, we evaluate the proposed ProgME framework, which has two major components — the programmable engine (FQAE) and the adaptive controller (MRT). We first look at the scalability and accuracy of FQAE and use two application scenarios to discuss the potential usage of FQAE in traffic engineering and security monitoring. We then discuss the memory cost and speed of MRT in identifying heavy hitters.

A. *Scalability of FQAE*

FQAE has a significant advantage in terms of scalability because it keeps per-flowset counters instead of per-flow counters. We perform empirical evaluation on the scalability of FQAE by comparing the number of counters one has to keep for flow-based and flowset-based measurement.

To understand the typical number of flows one should expect on high speed links, we look at the packet traces collected at OC-48 links by CAIDA [16] on April 24, 2003.

Configs	# flowsets	
	Log-Only (Orig/Disj)	All (Orig/Disj)
#1	19/22	40/55
#2	0/0	35/38
#3	0/0	800/845

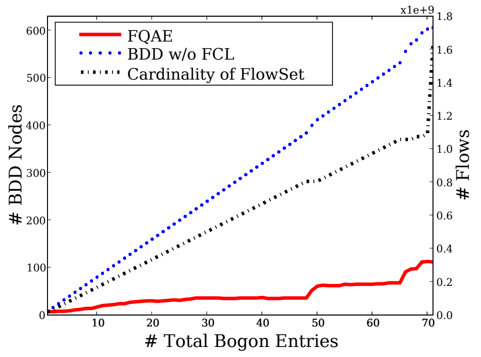
Table V: Size of Queries.

We choose to look at the 5-minute traces since 5-minute is a typical statistics report interval. As shown in Table IV, these trace files have a large number of flows (in the order of $10^5 - 10^6$) even when using simple flow definitions like source or destination IP address. If we use the two-tuple $\{sip, dip\}$ flow definition, the number of flows are even larger. We only present results from two traces. Other traces also have similarly large number of flows.

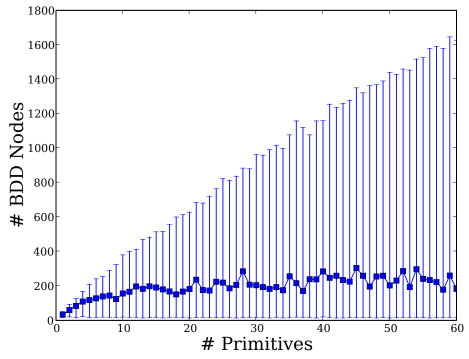
Since we have yet to see production usage of FQAE, we emulate the scenario of network administrator querying live traffic based on the production firewall configurations obtained from a tier-1 ISP and several campus networks. We use two approaches to emulate potential flowsets from these firewalls. In the first approach, we consider each "LOG" rule in the configuration file as a query for statistics. In the second approach, we consider every filtering rule as a query. Table V presents the number of user queries based on several firewall configurations. One can observe that the number of queries or disjoint sub-queries are significantly smaller than the number of flows one would observe from traffic traces. Note that such emulation does not fully utilize the capability of FQAE to reduce counters as we make a conservative assumption that every rule corresponds to a query.

One might argue that the number of independent flowsets generated by n user queries (denoted as m) can also be large. This is a legitimate concern since $m = 2^n$ in the worst case when every new flowset overlaps with all existing flowsets (Algorithm 1, line 12). However, we argue that the number of flowsets cannot be larger than the number of active flows. One can ensure that every flowset contains at least one active flow by aggregating flowsets with no active flows into one large flowset. Furthermore, our study on filter rules show that only a small portion of the rules overlaps with other rules. Consequently, the number of disjoint sub-queries (m) is only moderately larger than the number of queries (n) instead of being close to 2^m. This is also consistent with earlier study on firewall and router configurations. For a scenario of 300 queries with every 3 rules overlapping with each other, the number of disjoint flowsets one has to maintain counter for is just 700, which is significantly smaller than the number of flows on most high-speed links.

The reduced number of counters has multiple implications to the measurement architecture. First, it makes it possible to store the counters in the faster registers or SRAM. This is crucial for high-speed network devices. Second, it reduces the volume of data to be exported. Currently, Cisco NetFlow imposes a minimum five minutes interval between subsequent exports so that the mea-



(a) Bogons



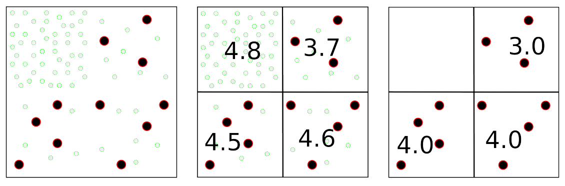
(b) Random Flowsets

Figure 10: Memory Cost of Flowsets.

surement data, probably coming from multiple vantage points in the network, will not overload the network. With the reduced number of counters, one can monitor the network at a higher temporal resolution and thus be more responsive to any anomalous events.

B. *Memory Cost of Flowsets*

Although FQAE can reduce the number of counters, a legitimate concern is that how much memory one has to spend for maintaining the underlying data structure of flowsets. To understand the memory cost associated with a flowset, we first look at a practical scenario of building a flowset representing all the bogon IP addresses in Figure 10a. We add each CIDR block in the current bogon list [17] one-by-one using the "union" operation and plot the corresponding number of BDD nodes required to represent this flowset ("FQAE"). For comparison, we plot the cardinality of the flowset and observe that the cardinality of the flowset increase significantly faster than the number of BDD nodes used to describe it. Furthermore, we plot the total number of BDD nodes used to describe each CIDR block. Without using FCL, a list representation of all the CIDR blocks ("BDD w/o FCL") will require more than five times the memory. Note that the last bogon entry is 224.0.0.0/3, which could match 2^{29} unique source IP address, thus resulting in the sharp increase in the cardinality of the flowset at the right end. However, its BDD representation requires only three nodes. In the case of using FCL to construct the flowset, only one additional node is required.



(a) Flows (b) Unbiased (c) Biased

Figure 11: Flow-based Estimation.

Figure 10b looks at the number of BDD nodes associated with randomly generated flowsets. On the X-axis, we vary the number of primitives used to define a flowset. Each primitive is defined by two-tuple $\{sip, dip\}$ with both mask lengths varying between 7 and 26. We randomly choose set operators ($\cap$, $\cup$, $\setminus$) to join the primitives so that the resulted flowset is non-empty. One can observe that the maximum number of nodes used to describe a flowset grows linearly with the number of primitives. This is consistent with our discussion in Section II-C and happens when the BDD representations of the primitives do not share common path. However, the average number of BDD nodes for a flowset grows significantly slower and is sub-linear to the number of primitives. Note that the memory required to define a flowset depends only on the primitives and the set operators and is independent of the traffic pattern it measures.

One might notice that we used the number of BDD nodes instead of the more direct *bytes* to evaluate memory consumption. This is because there are many BDD packages with node size varying between 8 and 36 bytes. Our current implementation is based on BuDDy [18], which uses 20 bytes per node. We believe that porting ProgME to another BDD package is easy and will not require any change to its algorithms. Please refer to [19] for a comprehensive survey on various BDD packages.

C. *Accuracy of FQAE*

The accuracy of measurement is of paramount concern for every management application. Existing flow-based measurement tools use *average* per-flow error as the primary measure of accuracy. Facing the challenge posed by the large number of flows, some recent research propose to keep counters preferentially for large or long-lived flows while excluding mice from occupying counters (more details in Section VI). Such techniques effectively produce biased flow statistics that results in smaller average error than unbiased random sampling. Note that unbiased random sampling naturally favor large flows because small flows have higher chances of not being sampled. Ideal unbiased per-flow statistics is only possible when every packet is counted. Please refer to [5] for comparison of different techniques in terms of average error achieved.

For a management application, lower average error does not always translate to higher accuracy for the answers to their queries, which directly affects their decisions and is more important. Consider the scenario depicted in

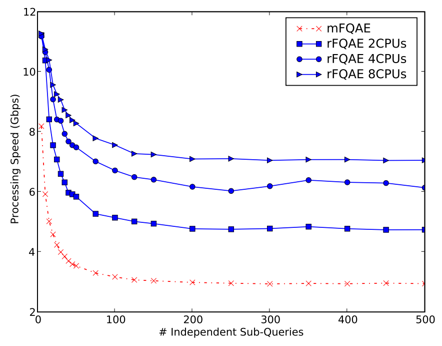


Figure 12: Speed of FQAE

Figure 11. The original traffic has some elephants and mice (Figure 11a). Figure 11b presents the ideal measurement results for the four queries on each quadrant if every single flow is recorded. Ignoring mice cause limited errors on queries dominated by elephants. However, for queries involving mostly mice (the top left quadrant in Figure 11c), the error can be very high.

Queries dominated by mice do have practical importance in network planning and monitoring. Typical examples are ICMP, DNS, and routing traffic. Although the absolute volume of these traffic might be small, their relative volume variation could be a useful indicator for administrators. Biased statistics that ignores these traffic is not suitable for monitoring mice-dominated traffic. More importantly, biased statistics makes it possible for attackers to evade the detection of volume-based monitoring tools. An attacker can generate a large number of flows, each with a different spoofed source IP address and small number of packets. Since these individually small flows are likely to be ignored by both biased and unbiased per-flow measurement, post-processing of statistics collected via traditional methods might fail to detect these attacks.

If there were no memory limitation and statistics could be maintained for every flow, per-flow statistics could achieve high per-query accuracy under any traffic condition. FQAE achieves the same effect by counting for each query directly. The analytical proof is skipped in this paper due to space limitation. Intuitively, If every single flow is recorded, it does not matter whether post-aggregation or pre-aggregation is used.

D. *Speed of FQAE*

In this section, we evaluate the speed of FQAE, which can help determine (1) if ProgME can handle modern link speeds, and (2) what can we do in cases when ProgME cannot match a link speed, e.g., change to faster processor or use sampling for load shedding. We run randomly generated queries on the OC-48 traces from CAIDA [16] to determine the average processing speed. All experiments are performed on a Dell computer with two quad-core Intel Xeon processor running at 2.00GHz.

From Figure 12, one can observe that the monolithic version of FQAE (*mFQAE*) achieves processing speed between 3 to 10 Gbps depending on the number of queries

	# Counters	Accuracy
Per-Flow	10^6	High
Elephants	10^3	Good
Superflow	1.7×10^5	High
FQAE	1	High

Table VI: Comparison on Deriving Traffic Demand.

to the systems. When there are more queries, FQAE is slower since it might need to match a packet through more queries. One might also notice that FQAE almost reaches a constant speed when there are more than 300 queries. This is because most packets have been matched by queries in the front part of each row (cf. TrafficSort in Section III-C). Additional queries at the end of each row only match a small number of packets, although they require more processing.

One can also find that the parallel version of FQAE (*rFQAE*) can effectively exploit the additional processing power to improve its speed. By enabling two CPUs, the processing speed almost doubles that of *mFQAE*. With all 8 CPUs enabled, *rFQAE* reaches about 7 Gbps speed.

E. *Case Studies on FQAE*

In this section, we present two case studies illustrating how administrators can use FQAE to accomplish their measurement goals. To use FQAE, one need to use FCL to specify the measurement queries. These two case studies show that it is not easy to produce the flowset definitions defining practical measurement tasks. We also compare FQAE with the following approaches:

1) *Per-Flow*: the ideal case that every flow is tracked.
2) *Elephant*: methods that produce biased statistics that favors heavy hitters, as discussed in Section V-C.
3) *Superflow*: In many routers and firewalls, it is possible to configure *LOG* rules to collect traffic statistics of superflows. LOG rules are similar to accept/drop rules except that its only operation is to increment the counter when a matching packet is observed. There is a large amount of research on packet classifiers, but for comparison purposes, we consider the most widely deployed variation, where a packet traverses through the rules sequentially until the first matching is found.

1) Deriving Traffic Demand: Our first task is to collect traffic statistics for deriving the traffic matrix of an ISP backbone. In particular, we consider the case that an administrator wants to measure the traffic going through a particular ISP with a list of AS number $Y_1, Y_2, \cdots, Y_n$. Such measurement has important application in traffic engineering and network planning.

The classical approach proposed by Feldmann et al. [20] is to perform per-flow measurement on an ingress router and then, based on the routing table at that moment, aggregate the flow statistics to find traffic demands. They also observed that around 1,000 elephants account for about 80% of the traffic demands. Therefore, techniques

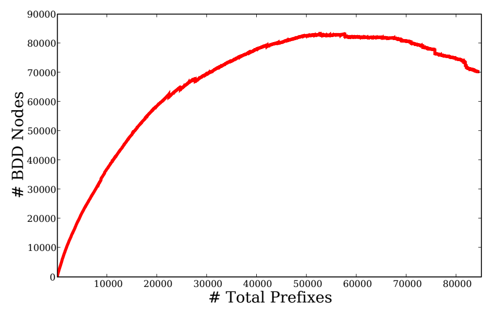


Figure 13: Memory of Prefixes

	# Counters	Accuracy	Computation/Pkt
Per-Flow	10^9	High	1 Hash
Superflow	71	High	35.5 match
FQAE	1	High	1 bdd_imp
Elephants	10^3	Low	×

Table VII: Comparison on Tracking Bogons.

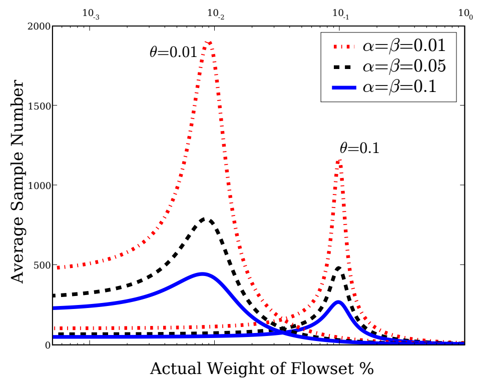


Figure 14: Average Sample Number $\mathbf{E}[N]$.

that ignore mice should still return sufficiently good statistics in general. One could associate a counter with every routing rule to collect statistics for each prefix (superflow) and then perform aggregation. However, since current BGP table carries around 170,000 prefixes [21], such approach will generate a large amount of data.

We propose a two-step approach to pre-process the routing table and compute the flowset that will go through AS Y. First, we process the routing table to find all prefixes that contains any of $Y_i, Y_2, \cdots, Y_n$ in its AS path. Then we use FCL to compute the union of the selected prefixes to find the flowset. We experiment by applying this approach on processing the BGP routing table dumped by the Route Views project [22] and find this approach viable. We identified 84,312 prefixes (among a total of 188,275 prefixes in the routing table) that might traverse through this tier-1 ISP (with 13 AS numbers) on its AS path. Figure 13 presents the BDD nodes used throughout our computation. The final flowset that represents the union of them requires a total of 70,291 BDD nodes (1.4MB using BuDDy and can be reduced to 560KB using other packages). At the beginning, adding new prefixes causes the BDD to require more nodes to enumerate more paths. However, the number of BDD nodes used to describe such flowset peak at about 80,000 nodes (56,000 prefixes) and decreases with more prefixes. This is because the large number of BDD paths actually present more opportunity for BDD to summarize the entire sub-tree into one node.

2) Tracking Bogons: The second task we consider is to track bogons, which are packets with reserved or unallocated source IP addresses. Packets from these addresses are often used by spammers or attackers and otherwise have no legitimate reason to appear on the Internet. Since these source IP addresses are spoofed, differentiating them is not meaningful. Administrators would normally want to track the aggregated volume of bogons as a single metric. Furthermore, administrators have to be prepared for the worst case since this is a security-oriented application.

The current bogon list [17] has 71 non-aggregated CIDR blocks (about 10^9 unique IP addresses). Keeping per-flow counter for these bogons is clearly unrealistic, even though it has high accuracy and requires only a single hash operation to derive the flow ID. Techniques that focus on elephants are not suitable here as we discussed in Section V-C. For superflow-based measurement, a packet will be compared with 35 bogon IP blocks on average. Using FQAE, one can pre-compute the union of all 71 CIDR blocks in bogon list as one flowset (as in Figure 10a). Consequently, one only needs to maintain one counter for all packets with bogon source.

F. Speed of MRT

In network monitoring, especially if it is security-related, it is important to detect a heavy hitter in the shortest time possible. In addition to the memory cost factor, the sample number required for the hypothesis to be conclusive (denoted as N) is another key parameter. For a flowset with weight F_w, the expected value of N ($\mathbf{E}[N]$) is a joint function of θ^+, θ^- F_w, α and β, as in Eq. 17.

$$\mathbf{E}[N] = \frac{L(F_w)\log B + (1 - L(F_w))\log A}{F_w \log \dfrac{\theta^+}{\theta^-} + (1 - F_w)\log \dfrac{1-\theta^+}{1-\theta^-}} \tag{17}$$

In Eq. 17, $L()$ is the operating characteristics (OC) function of the test. Directly evaluating $L(F_w)$ is difficult. Therefore, Wald [14] proposed a numerical method to evaluate $L(F_w)$, which we used here to calculate $\mathbf{E}[N]$.

Figure 14 presents $\mathbf{E}[N]$ under various scenarios. For a given α and β, Average sample number (ASN) is larger when F_w is close to the threshold θ and is small when the weight is either significantly larger or smaller than θ. This property presents a much desired feature for heavy hitter identification — heavier elephants will be identified faster than not-so-significant elephants.

One can use Figure 14 together with Eq. 16 to determine the expected speed of MRT in identifying elephants. Consider the case that we want to find all flows with weight larger than 0.01 and one flow f has a weight of 0.1. The worst case scenario is that f is the sole flow in the flowset that covers it. In this case, it takes an average of 26 samples for the hypothesis test to conclude that MRT should zoom into this flowset. For a two-tuple $\{sip, dip\}$ definition of flow and a memory cost factor of 256, it takes eight iterations, i.e., 208 packets to identify the ID of this flow. This is the worst-case and the actual speed of MRT depends on the traffic pattern and can be faster.

VI. Related Work

Cisco's Netflow V9 [23] and IETF's IPFIX [24] propose very generalized definition of flow. In RFC 5101, the definition of flow is essentially a set of packets selected based on functions applied to packet header fields, characteristics of packet or treatment of packet. It is however unclear what functions one can define. One could use FCL as the function to define flow in conformance to RFC 5101.

Online aggregation [25, 26] has received considerable attention in the database community. A typical example is to find the sum or average of a large number of objects. Instead of running through a large number of objects and return an accurate result after a long latency, such systems use statistical methods to provide running estimation so that users can decide in real time whether to continue. If a database of flow/packet records has been built, such a system can be adapted to query a database of flow records. The proposed flowset composition language (FCL) can be used for efficient specification of user queries, and FQAE can be used for aggregation on the database side.

There are several work on producing traffic summary or identifying hierarchical heavy hitters. Aguri [27] is a traffic profiler that aggregates small flow records (both temporal and spatial) until the aggregated weight is larger than a certain threshold. Autofocus [28, 29] is a traffic analysis and visualization tool that finds both uni-dimensional or multi-dimensional *clusters* from traffic trace or flow records. These tools requires per-flow statistics to make summary bottom-up. They are more engineered to work offline to find an effective presentation of traffic statistics but cannot improve the scalability of the measurement tools. There are some online variants that identify hierarchical heavy hitters without maintaining per-flow counters. Zhang et al. [30] applied packet classification algorithms dynamically (upon reaching a fixed threshold) to identify hierarchical heavy hitters top-down. The MRT algorithm in this paper also zoom into the heavy hitters top-down, but use SPRT to update the flowsets with proved optimality. Both offline and online aggregation are performed along hierarchies and driven entirely by traffic. They do not consider the different preference that administrators might have, e.g., to cluster traffic on port 80 with port 8000 instead of port 81.

In Section V-C, we compared FQAE with techniques that produce biased flow records to reduce resource consumption. This is exemplified by the work by Estan and Varghese [5], which calls to "focus on the elephants and ignore the mice" in flow statistics collection. They proposed two techniques, namely *sample and hold* and *multistage filter*, to achieve this goal. Of similar spirit are the work using smart sampling techniques. Threshold sampling [31, 32] is a stream-based method fits ideally for online monitoring. Priority sampling [33] follows the similar spirit of online aggregation and is more suitable for querying a database of flow records. Another class of elegant formulation are the coincidence-based techniques [34, 35] that exploit the fact that one is more likely to observe n consecutive packets from the same flow if the flow is large or long-lived. These techniques favor large flows without knowledge of user requirements, thus unsuitable when mice, e.g., DDoS traffic, are of interest. ProgME can complement these techniques by defining flowset that should receive preferential treatment, e.g., by setting different thresholds for different flowsets. ProgME can also use some of those techniques to improve its adaptive engine. For example, one could use coincidence-based techniques together with SPRT to improve the zooming process of the MRT algorithm.

Adaptive NetFlow (ANF) [36] is a scheme that dynamically adapts the sampling rate and the size of time bin in order to reduce the number of flow records while maintaining the accuracy. ProgME and ANF are complementary to each other since ProgME offers spatial adaptability while ANF achieves temporal adaptability.

Sangireddy and Somani [37] presented a high-speed routing lookup engine based on Field Programmable Gate Arrays (FPGAs). Their fundamental approach is to express packets going to a certain next-hop as a BDD and then process the BDD with the SIS package to generate the equivalent combinational logic, which can be mapped to FPGA. The performance of the hardware realization is reported to reach about 200 million lookups per second (varies according the routing table under consideration) – sufficient for 200Gb/s links. This work shows the feasibility for a similar hardware realization of ProgME on FPGA-based platform, which we explored separately [38].

Commercial products from Cisco [23], cPacket [39], and Gigafin [40] provide hardware-accelerated collection and processing of flow records or flow aggregate records. Unfortunately, we do not have sufficient information about their implementation details for a meaningful comparison.

VII. Conclusion and Future Work

In this paper, we presented ProgME, a framework for programmable network measurement. The key idea of ProgME is to collect traffic statistics based on a novel and versatile concept of *flowset* i.e., arbitrary set of flows, instead of the traditional inflexible concept of flow. The core of ProgME is a flowset-based query answer engine (FQAE), which can be programmed by users and applications via the proposed flowset composition language. Knowledge about user requirements offers measurement tools a fresh perspective and enables them to adapt itself by collecting statistics according to the tasks at hand. We further extended ProgME with an adaptive multi-resolution tiling (MRT) algorithm that can iteratively re-program itself to identify heavy hitters. We show that ProgME, being a versatile tool, can adapt to different measurement tasks.

In addition to the software implentation in this paper, we also explored implementing ProgME on a specialized

FPGA-based hardware platform [38]. Our implementation is based on a parallel and pipelined programmable architecture and assembles the row-level parallelism implementation in this paper. The core of this architecture is an programmable array of processing elements, each mapped to a flowset and providing mapping and counting function for the flowset. We believe this demostrates ProgMe's applicability to wider hardware platform.

References

[1] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: Towards Programmable Network MEasurement," in *Proc. ACM SIGCOMM*, 2007.
[2] N. Brownlee, C. Mills, and G. Ruth, "Traffic Flow Measurement: Architecture," RFC 2722, 1999.
[3] D. Plonka, "FlowScan: A network traffic flow reporting and visualization tool," in *Proc. USENIX LISA*, 2000.
[4] P. Phaal, S. Panchen, and N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks," RFC 3176, 2001.
[5] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Computer Systems*, 2003.
[6] D. Eppstein and S. Muthukrishnan, "Internet packet filter management and rectangle geometry," in *Symposium on Discrete Algorithms*, 2001.
[7] S. Hazelhurst, A. Attar, and R. Sinnappan, "Algorithms for improving the dependability of firewall and filter rule lists," in *Proc. Dependable Systems and Networks*, 2000.
[8] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A Toolkit for FIREwall Modeling and ANalysis," in *Proc. IEEE Symp. Security & Privacy*, 2006.
[9] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, 1986.
[10] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?" in *Proc. IEEE INFOCOM*, 2003.
[11] A. Feldmann and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proc. IEEE INFOCOM*, 2000.
[12] H. Hamed and E. Al-Shaer, "Dynamic rule-ordering optimization for high-speed firewall filtering," in *ACM Symposium on InformAtion, Computer and Communications Security*, 2006.
[13] S. Acharya, J. Wang, Z. Ge, T. F. Znati, and A. Greenberg, "Traffic-aware firewall optimization strategies," in *Proc. International Conference on Communications*, 2006.
[14] A. Wald, *Sequential Analysis.* John Wiley & Sons, 1947.
[15] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," in *Proc. IEEE Symp. Security & Privacy*, Oakland, CA, 2004.
[16] "CAIDA: Cooperative Association for Internet Data Analysis," http://www.caida.org/home/.
[17] Team Cymru, "The Team Cymru Bogon List v3.4," http://www.cymru.com/Documents/bogon-list.html, Jan 2007.
[18] J. Lind-Nielsen, "BuDDy version 2.4," http://sourceforge.net/projects/buddy, July 2004.
[19] G. Janssen, "A consumer report on BDD packages," in *Proc. IEEE Symp. Integrated Circuits and Systems Design*, 2003.
[20] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational IP networks: Methodology and experience," in *Proc. ACM SIGCOMM*, 2000.
[21] "CIDR report," http://www.cidr-report.org/, Jan 2007.
[22] "University of Oregon Route Views project," http://www.routeviews.org/.
[23] "NetFlow v9 Export Format," http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/123newft/123_1/nfv9expf.htm.
[24] "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," RFC 5101, 2008.
[25] J. M. Hellerstein, P. Haas, and H. J. Wang, "Online aggregation," in *Proc. ACM SIGMOD*, 1997.
[26] N. Alon, N. G. Duffield, C. Lund, and M. Thorup, "Estimating sums of arbitrary selections with few probes," in *PODS*, 2005.
[27] K. Cho, R. Kaizaki, and A. Kato, "Aguri: An aggregation-based traffic profiler," in *Proc. Quality of Future Internet Services*, 2001.
[28] C. Estan, S. Savage, and G. Varghese, "Automatically inferring patterns of resource consumption in network traffic," in *Proc. ACM SIGCOMM*, 2003.
[29] "Autofocus," http://ial.ucsd.edu/AutoFocus/.
[30] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *Proc. Internet Measurement Conference*, 2004.
[31] N. Duffield and C. Lund, "Predicting resource usage and estimation accuracy in an IP flow measurement collection infrastructure," in *Proc. Internet Measurement Conference*, 2003.
[32] N. G. Duffield, C. Lund, and M. Thorup, "Learn more, sample less: control of volume and variance in network measurement," *IEEE Trans. Information Theory*, vol. 51, 2005.
[33] ——, "Flow sampling under hard resource constraints," in *Proc. ACM SIGMETRICS*, 2004.
[34] M. Kodialam, T. Lakshman, and S. Mohanty, "Runs bAsed Traffic Estimator (RATE): A simple, memory efficient scheme for per-flow rate estimation," in *Proc. IEEE INFOCOM*, 2004.
[35] F. Hao, M. S. Kodialam, T. V. Lakshman, and H. Zhang, "Fast, memory-efficient traffic estimation by coincidence counting," in *Proc. IEEE INFOCOM*, 2005.
[36] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," in *Proc. ACM SIGCOMM*, 2004.
[37] R. Sangireddy and A. K. Somani, "High-speed ip routing with binary decision diagrams based hardware address lookup engine," *IEEE J. Selected Areas in Communications*, 2003.
[38] F. Khan, L. Yuan, C.-N. Chuah, and S. Ghiasi, "Programmable and real-time network traffic measurements," in *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.
[39] "cPacket Networks," http://www.cpacket.com/.
[40] "GigaFin Networks," http://www.gigafin.com.

Lihua Yuan (lyuan@microsoft.com) is currently a Research Software Development Engineer at Microsoft. He received his Ph.D. in Electrical and Computer Engineering from the University of California, Davis in 2008. His research interests are in the area of computer networks and distributed systems, with a focus on network management, measurement, and security.

Chen-Nee Chuah (chuah@ece.ucdavis.edu) is a Professor in the Electrical and Computer Engineering Department at the University of California, Davis (UCD). She received her Ph.D. in Electrical Engineering and Computer Sciences from the University of California, Berkeley in 2001. Her research interests are in the area of computer networking and distributed systems, with an emphasis on Internet measurements, network management, anomaly detection, and online social networks.

Dr. Prasant Mohapatra (prasant@cs.ucdavis.edu) is currently a Professor in the Department of Computer Science at the University of California, Davis. Dr. Mohapatra received his Ph.D. in Computer Engineering from the Pennsylvania State University in 1993. He was/is on the editorial board of several IEEE- and ACM-sponsored journals and conferences. Dr. Mohapatra's research interests are in the areas of wireless networks, sensor networks, Internet protocols and QoS.