A Generic Language for Application-Specific Flow Sampling

Harsha V. Madhyastha
University of Washington
harsha@cs.washington.edu

Balachander Krishnamurthy
AT&T Labs–Research
bala@research.att.com

ABSTRACT

Flow records gathered by routers provide valuable coarse-granularity traffic information for several measurement-related network applications. However, due to high volumes of traffic, flow records need to be sampled before they are gathered. Current techniques for producing sampled flow records are either focused on selecting flows from which statistical estimates of traffic volume can be inferred, or have simplistic models for applications. Such sampled flow records are not suitable for many applications with more specific needs, such as ones that make decisions across flows.

As a first step towards tailoring the sampling algorithm to an application's needs, we design a generic language in which any particular application can express the classes of traffic of its interest. Our evaluation investigates the expressive power of our language, and whether flow records have sufficient information to enable sampling of records of relevance to applications. We use templates written in our custom language to instrument sampling tailored to three different applications—BLINC, Snort, and Bro. Our study, based on month-long datasets gathered at two different network locations, shows that by learning local traffic characteristics we can sample relevant flow records near-optimally with low false negatives in diverse applications.

Categories and Subject Descriptors

C.2.3 [**Communication Networks**]: Network Operations — *Network monitoring*; D.3.3 [**Programming Languages**]: Language Classifications — *Specialized application languages*

General Terms

Algorithms, Experimentation, Languages, Measurement

Keywords

Flow sampling, Application-specific traffic monitoring, Language design

1. INTRODUCTION

Routers can be configured to output summaries of traffic flows providing aggregate information about a flow of packets between a pair of endpoints. Such summaries, commonly referred to as flow records, consist of information about each flow in the form of duration, number of packets, number of bytes, etc.

Due to increasing traffic volumes on the Internet, routers and flow collectors are unable to keep up with maintaining flow information and exporting records for all traffic. Routers/flow collectors are now configured to output records for only a sampled subset of traffic. The sampling can be simply statistical, such as 1-in-n, for various values of n anywhere from 10 to 500. Or it can be *smart* [12]—biased to the percentage of traffic in applications.

Several solutions [15, 19, 12] have been proposed for tuning the sampling algorithm. However, the focus of sampling has largely been limited to exporting flow records from which unbiased statistical estimates of traffic volume can be inferred. While these techniques are clearly novel, no single sampling technique will suffice to meet the needs of various applications. For example, an application maybe interested in a specific traffic subset such as DNS.

There has also been work [32] towards tailoring the sampling of flow records to the needs of specific applications. Systems that enable application-specific traffic monitoring [4, 6] are also relevant in this context. However, all of these solutions have simplistic application models; the traffic of interest to an application has to be specified either as an SQL-like query, or as a conjunction of logical predicates. Such models do not suffice for more complex applications that make decisions across flows, such as detecting denial of service (DoS) attacks in combination with SNMP data [3, 30].

To bridge this disconnect between application requirements and the sampling algorithm employed in routers, we present a language for specifying the traffic of interest to applications. Devising and implementing a new sampling algorithm for every application would be cumbersome. Instead, it is desirable to have an application-independent implementation of a sampler that takes a template of rules as input from any particular application. This template will express the application's traffic of interest, and will be used to sample accordingly. In designing a language for templates, we walk the tightrope of making the language general-purpose to express the diverse needs of applications, and yet keeping it simple enough to be executed on routers and flow collectors; we trade off completeness for simplicity. We design a language that can be compiled into lightweight constructs such as finite state machines, bitvectors, and hash tables.

The contribution of our work can be seen in light of the spectrum spanning applications and current solutions for application-specific traffic monitoring. At one end of this spectrum are applications themselves, each of which can have an arbitrarily complex flow of logic. At the other end of the spectrum are existing solutions [32, 4, 6] for application-specific sampling of flow records, which model applications as simplistically as possible so as to minimize the resource utilization in emulating their execution. Our work is at the middle of this spectrum—we aim to capture most of the application's logic in an application-generic manner, but potentially at the expense of compromising on resource utilization.

In our quest to build a sampler that has information about application needs, we investigate whether flow records contain sufficient information to enable sampling of records of relevance to applications.

- We gathered unsampled flow data for a month each at two different network locations, and used three different measurement-related network applications (Snort [28], Bro [25], and BLINC [17]) as exemplars.
- Our custom template language was expressive enough to characterize the diverse interests of these applications.
- For each application, we sampled flows based on its templates. We then evaluated the accuracy of sampling by comparing these sampled flows with the flows chosen by normal execution of the application. By incorporating local traffic information into the input templates, we obtain near-optimal sampling in a large number of cases.

Though we focus on sampling flow records in this paper, our language is applicable to generic records.

2. LANGUAGE FOR TAILORED SAMPLING

Flow records obtained using deterministic and random sampling of packets may prove less useful for applications other than volume estimation. 1-in-n sampling or smart sampling can generate flow records that provide good aggregate statistics. However, for an application interested in a specific subset of flows, it would be more beneficial to selectively have more complete information about the flows of its interest.

Applications have a better idea of traffic of interest than routers or flow collectors do. Applications have rules to identify traffic of interest, e.g., an Intrusion Detection System (IDS) has a set of rules to identify attack traffic. If routers and flow collectors are made aware of such rules, they could selectively gather records only for flows that are of interest to the application using these flow records.

A key requirement for application-specific sampling is that the classes of traffic of interest to the application be supplied as input to the sampling module. These traffic classes need to be specified as a template comprising simple rules that require the sampler to test predicates based on the fields present in flow records. The language in which the template is specified needs to be application-generic to avoid reimplementing the sampling engine for each application that can benefit from tailored sampling. Constructing a template is a one-time task, so the effort spent in writing the template can be leveraged across all network locations that implement sampling tailored to the same application.

We design a template language with the goal of enabling network administrators to characterize the traffic of interest to a wide range of applications. We have an initial implementation of a sampler that matches templates in this language with flow records, and we are currently working on optimizing its resource utilization.

Two stages of sampling occur in the process of generating flow records. Routers sample the packets they receive in computing summaries of flows. Flow collectors sample the summaries they receive from routers before writing them to disk. In our work, we handle only flow record sampling at the flow collector, and assume the generation of flow records at the router without any packet sampling. Hence, our template language is to be matched with flow records, and our sampler that performs this matching runs at the flow collector. Incorporating packet sampling into our framework is imminent future work.

2.1 Modeling applications

We model applications as receiving a stream of network data records as input, ranging from application-level records to flow records to packet streams. Each application is characterized by two functions: *state* (capturing current internal state of the application) and *select* (a boolean function that determines records of interest).

field	flow record element (e.g., *src_addr*, *dst_port*)
value	a field's value (e.g., *192.168.10.5*, *80*)
operator	a boolean operator (e.g., =, !=, =∼, ∼=)
predicate	a (field, value) comparison (e.g., *src_addr = 192.168.10.5*, *dst_port != 80*)
clause	conjunction of predicates
statement	disjunction of clauses
block	set/sequence of statements

Table 1: Template language specification terminology

Record-triggered applications: First, we consider applications where *select* is applied on each new data record received by the application. *select* takes two arguments—the current record, and the current state returned by *state*, which is a function of all the records received by the application prior to the current record. *state* is first executed to update the current state, and then *select* is executed to determine if the current record is of interest based on the new state. An example application is identifying out-of-order TCP packets in a packet stream. *state* captures the current state of each TCP connection, and *select* returns true whenever a packet is deemed to be out-of-order based on the current state.

Time-triggered applications: The second class of applications is where *select* is executed only at certain scheduled points in time. Such applications use all (or a subset of) records that arrive to build up state via *state*. At the end of each period, *select* is executed on each stored record along with the current state as an argument to see if it is of interest. An example application is one that parses a Web server log and identifies all objects fetched by a client that downloaded more than 1GB from the server in a day, as a way of identifying an unusual client. Here, *state* stores the cumulative number of bytes downloaded by each client since the beginning of the day. *select*, executed once a day, returns true for each object fetched by a client whose cumulative total over the day is greater than 1GB.

2.2 Language Specification

Our design of the template language (terminology in Table 1) is based on the above two broad classes of applications. The language primitives are predicates that test a field present in flow records against a value using simple boolean operators, such as equals (=), not equals (!=), belongs to (=∼), and contains (∼=). These primitives can be combined to specify a variety of application characteristics.

In record-triggered applications, a record is chosen by the *select* function based on the current state—we seek to capture the subsequence of records in the data stream responsible for this state. To detect out-of-order TCP packets, the selection of the current packet only depends on the packets received until now for the *same* 5-tuple as the current packet.

To enable sampling of a flow record, the template needs to identify the format of the record and the relevant prior subsequence of records. We model the format of a flow record by means of statements that are compositions of predicates in the disjunctive normal form (DNF), i.e., a disjunction of clauses, each of which is a conjunction of predicates. For example, the statement

((src_addr=192.168.10.5 and dst_port=80) or
(src_addr=192.168.10.5 and dst_port=22))

matches all records for flows from the host *192.168.10.5* to either port 80 or 22. A sequence of such statements (a *block*), identifies the current record and the prior subsequence of records in the

flow stream that cause *select* to return true. The build up of the current state that triggers *select* might also depend on a particular set of records being received prior to the current record, not necessarily in sequence. For example, connection state is characterized by a subsequence of packets whereas the count of number of bytes from a host is characterized by a set. Hence, our language models record-triggered applications by means of two kinds of blocks—sequential and unordered—which we map to finite-state machines and bit vectors. A sequential block is matched when all its constituent blocks/statements are matched in order, whereas the constituents of unordered blocks can be matched in any order.

Our language also permits predicates to match fields with variables, e.g., *((src_port=P and dst_port=P))* matches all flow records with same source/destination port without requiring a value, as it may not be known *a priori*.

We next consider the class of time-triggered applications. Since the *select* function in such applications is invoked only periodically, the application's state that triggers the selection of a record cannot be modeled as either a set or sequence of records. Instead, we have to model the application's state. In general, the data structures used by the application to store its state could be arbitrarily complex. Our abstraction is a set of hash tables, the schemas for which will be specified by the template. When the end of a period is triggered, a boolean *if* clause on the values stored in these hash tables models the selection of a record by *select*.

The template for time-triggered applications is specified by means of timed blocks: a set of statements each of which is associated with a hash table. The hash table associated with a statement is updated whenever a flow record matches the statement. The tuple of variable assignments in each statement serves as the key for the hash table associated with the statement, e.g., a hash table associated with the above statement, that checks for equality of source and destination ports, would have its key as the value assigned to the variable P. In each hash table, the sampler maintains two kinds of aggregate values. First, for each key, it maintains counts of various properties of flow records that match the statement with that key, e.g., the number of source addresses, the number of destination ports, the number of flows. Second, it maintains aggregate values across keys, e.g., maximum number of flows that matched any single key. We choose to store these values in each hash table since we believe such values are the ones widely used in measurement applications.

To model the *if* clause that would be executed at the end of a period, each statement is associated with an auxiliary statement—a DNF composition of predicates that test the values in the hash table associated with the statement. An example of a statement in a timed block is *((src_port=S and dst_port=80)) with [[num_bytes* $\geq 10^9$ *]]*, which is matched for all records to port 80 for hosts that exchanged more than 1GB of traffic with this port. Each timed block consists of a set of such statements associated with auxiliary statements. At the end of a period, a block is declared to have been matched if the auxiliary statements associated with every statement of the block are satisfied.

We present example templates from the applications we consider in our study. Figure 1 shows slightly simplified versions of portscan templates in Bro and BLINC. Bro's template is based on sequential and unordered blocks as it is a record-triggered application. Blocks can be nested arbitrarily, forming a hierarchy of blocks, with all leaf blocks containing statements. The block at the root of this hierarchy is the *root* block. In this example, Bro first detects candidates for port scan detection as hosts that contact 3 distinct ports within the span of 30 seconds. Bro then flags a port scan when an identified candidate contacts 3 distinct ports on the same destination

```
match sequential_block {
    match sequential_block {
        ((src_addr=S and dst_port=P1 and time=T));
        ((src_addr=S and dst_port=P2 and time∼=[T, T+30]));
        ((src_addr=S and dst_port=P3 and time∼=[T, T+30]));
    }
    match unordered_block with timeout=900 {
        ^ ((src_addr=S and dst_port=Q1 and dst_addr=D));
        ^ ((src_addr=S and dst_port=Q2 and dst_addr=D));
        ^ ((src_addr=S and dst_port=Q3 and dst_addr=D));
    }
}
where (S!=∼[123.45.0.0/16, 68.0.0.0/8]) and (D=∼[68.0.0.0/8])
```

(a)

```
match ((proto=6)) timed_block [300] [flush 300] {
    ^ ((src_addr=S and dst_port=P and failed=1));
    ((src_addr=S and dst_port=P and failed=0)) with [[num_flows ≤ 4]];
    ((src_addr=S and failed=1)) with [[num_dst_ports ≥ 50]];
}
```

(b)

Figure 1: Templates for portscan in (a) Bro (b) BLINC

within the next 900 seconds. The specifications of these timeouts of 30 and 900 seconds is enabled by associating unordered and sequential blocks with a timeout value indicating when the sampler can throw away the state maintained in matching this block. Note that the application itself will need to be aware of such timeouts since it too needs to garbage collect its state. When the root block in a template is matched, all the flow records that matched statements marked with the "^" tag are sampled. The use of variables S, $P1$, D, etc. enable this template to detect port scans without specifying the actual source or destination addresses or the particular port numbers probed.

BLINC's template for port scan uses a timed block. The argument *[300]* associated with the block indicates the period with which the auxiliary statements in the block need to be checked. The *[flush 300]* argument indicates the period with which the hash tables associated with each statement need to be cleared. Until the end of a period, the hash tables associated with each statement in the block are updated for every record that matches the statement. At the end of the period, the auxiliary statements associated with each statement are checked. If all are satisfied then the records that matched the statements marked with the "^" tag are sampled. In this template, all records for failed connections (indicated by the absence of packets with payload in the flow) from a particular host to a particular destination port are sampled if the number of successful connections between this (host, port) pair was not greater than 4, and the host had failed connections to more than 50 ports. The *((proto=6))* statement after the keyword *match* serves as a quick check to prune out uninteresting flow records. Beyond these examples, an unordered block can be associated with a threshold k for the number of its constituents to be matched, and the state associated with a block is discarded if one of its statements associated with a negation tag $\sim$ is matched.

3. EVALUATION

We evaluate two aspects of application-specific sampling: Is our template language expressive enough to characterize popular applications? And do traffic summaries in flow records have sufficient information to select records of relevance to applications? The absence of payload-dependent information in flow records will limit the ability to map application criteria that depend on the contents of packets. We consider two popular classes of applications—

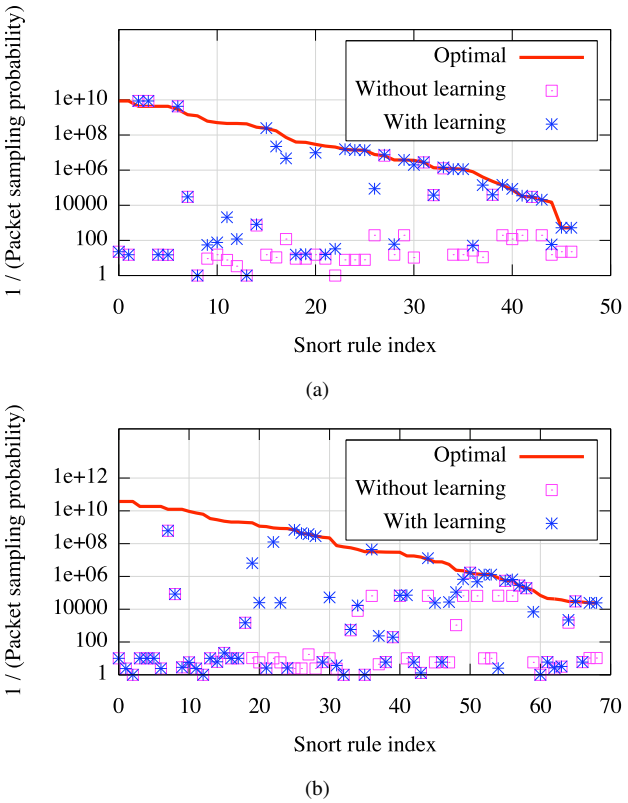


Figure 2: Sampling probabilities ($\frac{no.\ packets\ in\ sampled\ flows}{total\ no.\ packets}$) obtained on the first half of (a) *Corp* and (b) *Univ* with Snort's templates as input. Note log-scale on y axis. Inverse of sampling probability is plotted. Higher is more compact.

intrusion detection and traffic classification. We examine Snort [28] and Bro [25] as representative IDSes as they constitute opposite ends of the IDS spectrum with respect to state maintenance. The traffic classification application we consider is BLINC [17]. IDS applications have value primarily at the edge of the network, whereas traffic classification is relevant both in the core and at the edge.

3.1 Measurement Datasets

We gathered data for a month at a corporate site *Corp* and a university *Univ*, both with hundreds of users. At both locations, we examined all traffic traversing a particular router. *Corp* was gathered at an OC-3 link which saw bidirectional traffic to the site; *Univ* is based on traffic from one of the links connected to a router at the edge of a campus network. We gathered the complete set of flow records (unsampled), and connected a second machine to a span port on the router to sniff all traffic going through the link. We ran Snort and Bro on this sniffed traffic. These results and all flow records were sent to an analysis machine where we ran BLINC on the flow records. We then performed sampling on this analysis machine and compared the sampled records with the packets/flows chosen by Snort, Bro, and BLINC. The optimal sampling is obtained when precisely those packets/flows of interest to the application are sampled, and this is our point of comparison.

3.2 Snort

Snort [28] is an open-source IDS that monitors networks by matching each packet it observes against its set of rules. A Snort rule is a boolean formula composed of predicates that check for specific values of various fields present in the IP header, transport header, and payload of packets. For each Snort rule, the template contains a single sequential block which comprises a single statement. This statement is the conjunction of all predicates in the Snort rule ignoring ones that test fields absent in flow records, e.g., packet payload. Also, to ensure a flow record that matches this statement is sampled, the statement is marked with the "^" tag.

Such a construction of the template ensures that whenever a packet matches a Snort rule, the record for the flow corresponding to this packet is sampled. To determine the sampling probability, we consider a hypothetical scenario where Snort examines only the packets in the sampled flows. The effective sampling probability obtained is the ratio of the number of packets in the sampled flows to the total number of packets in the traffic stream. Figure 2(a) compares this sampling probability for the 47 distinct Snort rules seen in either half of *Corp*, with the optimal sampling probability obtained by use of an oracle sampler that samples precisely those packets that trigger Snort alerts. Figure 2(b) does the same for the 69 Snort rules seen in either half of *Univ*. The curve plots the optimal sampling probability for each Snort rule in the order of increasing number of alerts. The squares plot the inverse of the packet sampling probability obtained if Snort were to only examine packets in the sampled flows. First, since our construction of templates corresponding to Snort rules ensures the absence of false negatives, the sampling probability yielded is always greater than the optimal sampling probability. Second, there are a few Snort rules for which optimal sampling is achieved. These cases correspond to Snort rules that test only fields present in flow records, e.g., a few rules detect anomalous traffic by only checking for the values of ICMP code and ICMP type, not the packet payload.

However, there are several rules that test the packet payload, for which the sampling obtained is significantly below optimal. To bridge this semantic gap between Snort's rules and the corresponding templates, we explore whether flows of interest to Snort exhibit any features not already specified in the Snort rules. The features learnt are useful and applicable only if valid beyond the learning phase. Therefore, we learn traffic characteristics from the first half of the dataset, modify templates based on the learning phase, and evaluate the resultant sampling on the second half.

To detect additional criteria, which we call *filters*, corresponding to each Snort rule, we correlate every alert fired due to a rule with all the flow records sampled with the equivalent template. The flow record corresponding to a particular alert is identified as the one whose sextuple [1] matches that of the packet which generated this alert and whose duration envelops the time of occurrence of this alert. Based on this correlation, the set of flow records sampled for each Snort rule is partitioned into two classes—correct matches (*CM*) and false-positives (*FP*).

We next determine for each Snort rule a filter that distinguishes the correct matches from the false positives associated with this rule. A generic system that can be employed for the detection of such filters is AutoFocus [14]. AutoFocus takes a set of flow records as input, and determines traffic clusters that account for large fractions of the traffic. We employ a simpler filter detection methodology on our datasets.

Our methodology has two steps to infer predicates as part of the filter—inferring predicates based on the equality operator and those based on inequalities. In the first step, we consider the following fields present in netflow records—source address and port, destination address and port, number of bytes, number of packets, and

[1]Sextuple of a flow comprises its 5-tuple (src addr, src port, dst addr, dst port, protocol), and type of service.

flow duration, and a synthesized field—average number of bytes per packet. For each of these fields, we determine whether there exists any particular value that distinguishes the correct matches from the false positives. We determine the value for each field that exists in at least 90% of the flows in *CM*. For example, all flows in *CM* could have destination port equal to 443, or more than 90% could have exactly 3 packets. We initialize our filter for each Snort rule as the conjunction of such *(field = value)* predicates identified. Since each of these predicates is valid for at least 90% of the flows in *CM*, their conjunction too will be valid for a large fraction of these flows, though not necessarily for more than 90%.

In the second step, we examine the four fields for which inequality predicates are applicable: number of bytes and packets, flow duration, and average number of bytes per packet. We consider those fields for which *no* predicate is obtained in the first step. For each such field, we determine whether there exists any threshold value that distinguishes the flows in *CM* from those in *FP*. We perform a binary search over the set of values applicable to each field. For each value, we determine the fraction of flows in the *CM* and *FP* sets, for which the field is greater/lesser than this value. If there exists a value for which the fraction in *CM* is greater than that in *FP* by at least 75%, we use that value as a threshold in our filter. For example, 95% of correctly matched flows could last for more than 10 seconds with only 10% of false positives lasting as long. For every field for which such a threshold is identified, a *(field* $\geq$ *threshold)* or *(field* $\leq$ *threshold)* predicate is added to the filter.

Both the thresholds we use, 90% for equality predicates and 75% for inequality predicates, were chosen based on our datasets. Either threshold can vary from 100% to 0%. The higher the threshold, the better the quality of the inferred filter. We slid both thresholds down from 100% in decrements of 5%, and determined that 90% and 75% where the highest values for these thresholds at which filters were detected for many rules. Sliding either threshold further down by another 5−10% provided only marginal utility in detecting more filters, while reducing the quality of the inferred filters.

Inference of filters based on very few occurrences will, however, not be statistically sound. Hence, we computed a filter for only those rules that were fired at least 20 times, the knee in the distribution of the number of times each Snort rule was triggered in the first halves of *Corp* and *Univ*. The set of filters inferred differed across the two sites, reflective of the local traffic characteristics at either site. The template for each Snort rule was then modified to include the conjunction of the predicates it already contained with the predicates in the filter inferred for this rule.

The asterisks in Figures 2(a) and 2(b) plot the inverses of the sampling probabilities obtained on flows in the first half of *Corp* using the modified templates as input. Compared to our initial results (the squares in Figures 2(a) and 2(b)), we see significant improvement. Near-optimal sampling is achieved for a large fraction of Snort rules, and even among those farther away from the optimum, several attain a sampling probability close to 1-in-100. The rules for which the sampling probability does not decrease much are predominantly the ones which were triggered less than 20 times, and hence, had no filter inferred. Note that with the addition of filters, the sampling probability can even be lesser than the optimum since filters are not perfect and can introduce false negatives. However, we find that 85% of Snort rules in *Corp* and 91% in *Univ* see less than 10% false negatives, with half in either dataset seeing none.

We studied the prevalence over time of our inferred filters, by using the filter-augmented templates to sample flow records in the second halves of *Corp* and *Univ*. The asterisks in Figures 3(a) and 3(b) show the inverse of the sampling probabilities obtained. Though the sampling probability is higher than that obtained in the

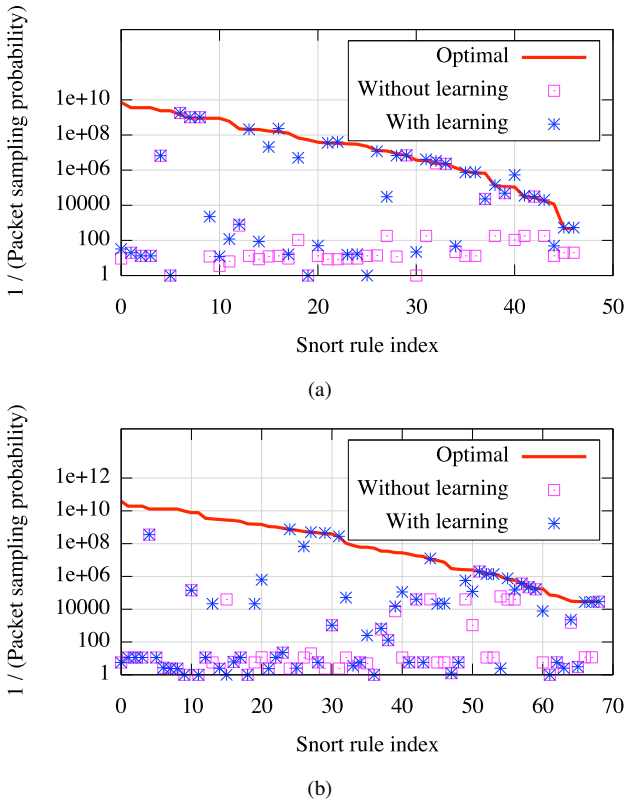


Figure 3: Sampling probabilities obtained on the second half of (a) *Corp* and (b) *Univ* with Snort's templates as input. Note log-scale on y axis.

first half of the dataset for several of the Snort rules, it is still significantly better than our initial results (the squares in Figures 3(a) and 3(b)). In fact, the sampling continues to be near-optimal for many of the rules. The distribution of false negative ratios has a longer tail than before, with even an instance of all the alerts being missed for a particular Snort rule. The large majority (78% in *Corp* and 83% in *Univ*), however, continue to have false negative ratios of less than 10%. More complex filter detection algorithms such as AutoFocus may help improve these numbers further. But, our simpler set of steps for detecting filters seems to work pretty well on our datasets.

To understand *why* we were able to detect such effective filters, we examined by hand the filters derived by our automated methodology described above. A classification of the different kinds of filters we observed is as follows (presented in decreasing order of the number of Snort rules for which such filters were derived):

Attack Probe Information: Many alerts involve the use of a carefully crafted train of packets that exploit a particular bug in some system. The Snort rules for such attacks contain signatures for the payload used in these attack probes. Since payload information is absent in flow records, the templates corresponding to such rules can lead to the sampling of a large number of false positive flow records. However, for many such attacks, the number of probes and the aggregate size of the probes is unique to attack flows, as compared to legitimate flows served by the targeted system. So, predicates that test for particular values of the number of bytes, the number of packets or the average number of bytes per packet are inferred to reduce the number of false positives.

Alert		1 / (Optimal Sampling Prob.)	1 / (Achieved Sampling Prob.)	False Negative Ratio
AddressScan / PortScan		8358	7980	5.7%
Sensitive Connection	First Half (w/o learning)	$7.8 \cdot 10^6$	371	0%
	First Half (w/ learning)		$7.2 \cdot 10^6$	1.6%
	Second Half (w/o learning)	$9.9 \cdot 10^6$	459	0%
	Second Half (w/ learning)		$1.0 \cdot 10^7$	4.1%

(a)

Alert		1 / (Optimal Sampling Prob.)	1 / (Achieved Sampling Prob.)	False Negative Ratio
AddressScan / PortScan		79485	64282	6.3%
Sensitive Connection	First Half (w/o learning)	$1.1 \cdot 10^7$	5523	0.04%
	First Half (w/ learning)		$8.1 \cdot 10^6$	8.9%
	Second Half (w/o learning)	$6.4 \cdot 10^6$	4700	0.2%
	Second Half (w/ learning)		$4.0 \cdot 10^6$	10.1%

(b)

Table 2: Results of sampling flow records in (a) *Corp* and (b) *Univ* given Bro's templates as input.

Attack Duration: For several Snort rules, the anomalous flows they identify last for extremely short periods of time or contain a small number of packets in comparison with legitimate traffic destined to the service specified in the rule. Predicates that test for thresholds on the number of packets or on the duration of the flow can be inferred for such rules.

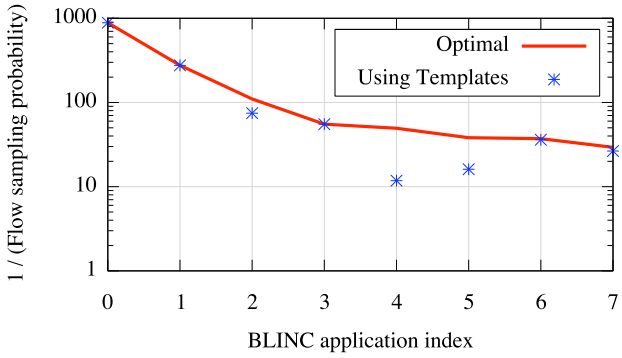
Absent Service: When the service targeted by a particular attack is not running locally, all traffic destined to this service is anomalous; there is no legitimate traffic. The filter inferred for a rule that detects such traffic has a single predicate checking if the destination port of the flow is equal to the port associated with this service. However, since all flows to this destination port are anomalous, the accuracy with which this alert is detected is high even without filtering.

Specific Attacker/Target: In our datasets, attacks corresponding to some of Snort's rules were observed to be either mostly launched from the same source host or largely targeting the same destination host. The specific attacker or target was recognized in these cases by inferring predicates that test for values of the source or destination IP.

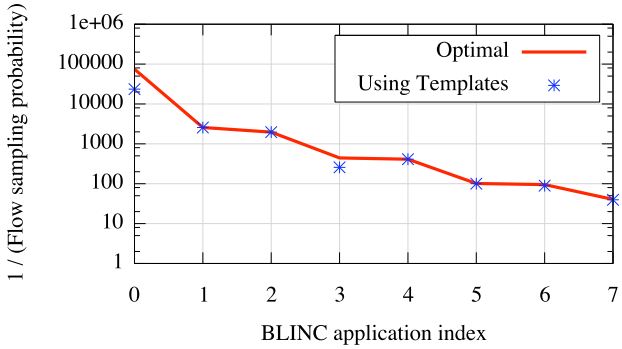
To better illustrate the inference of filters, consider the Snort rule that triggers the *ICMP digital island bandwidth query* alert. This rule checks for the presence of a particular string in the payload of ICMP packets. Since flow records do not have payload information, all flows corresponding to ICMP traffic match with the template corresponding to this rule. However, in the first half of *Corp*, our filter inference procedure detected that all flows corresponding to packets that triggered this alert had exactly one packet of size 1420 bytes. Use of this filter in sampling the second half of *Corp* helped reduce the sampling probability corresponding to this Snort rule from 1-in-100 to 1-in-5×10^6. Such information is not part of the Snort ruleset and varies across sites. Hence, to obtain more effective sampling, each site will need to learn these additional filters to reflect its local traffic characteristics.

3.3 Bro

Bro is another open-source system that monitors traffic at some central point in the network, and takes policy scripts as input from network administrators to define anomalous or malicious traffic. Bro maintains comprehensive state information about each connection to reduce false-positive alerts of Snort (which inspects each packet in isolation).



(a)



(b)

Figure 4: Sampling probabilities obtained on (a) *Corp* and (b) *Univ* with BLINC's templates as input. Note log-scale on y axis.

We observed three different types of Bro alerts in either dataset. Alerts detected in Bro's *weird.bro* policy script are excluded from our analysis, since these correspond to oddities in traffic, such as inconsistencies in IP fragments or corrupt TCP options, which are largely inconsequential. The other two classes of Bro alerts are address/port scans, and sensitive connections; detected by the policies in the *scan.bro* and *hot.bro* scripts. We constructed templates corresponding to the alerts detected in these scripts. As Bro is stateful, our templates for Bro are based on sequential and unordered blocks, (c.f. example template for PortScan in Section 2). The different statements in the template identify various records that Bro looks for before flagging some traffic as being abnormal or malicious. Like Snort, our templates for Bro are built with the assumption that any predicate that tests fields absent in flow records is true.

Table 2 shows the results of sampling flow records in both *Corp* and *Univ* with Bro's templates as input. The sampling achieved for address/port scans is near-optimal, and so we do not consider either half of the dataset separately. Scanning behavior is largely detected based on flow-level properties, such as the number of destination hosts/ports contacted by a host.

On the other hand, the sampling obtained in the case of sensitive connections is significantly below optimal in either half of the dataset, due to the information loss in flow records as compared to a packet stream. However, on learning additional filters by a methodology similar to that used with Snort, near-optimal sampling is obtained in both halves of either dataset, with minimal false negatives.

3.4 BLINC

BLINC [17] is a system for traffic classification developed recently, that takes a set of flow records as input and determines

the application responsible for each flow without examining packet payload or knowing port numbers used by various applications. We drew up templates corresponding to the *graphlets* used by BLINC to characterize applications, and used these templates to sample records in either dataset.

Figure 4 compares the sampling ratio obtained for each application identified by BLINC, with the corresponding optimal sampling ratio. The sampling is near-optimal for 6 of the 8 applications. The sampling is not as close to optimal for the other 2 applications, games and address scans, because BLINC employs some optimizations to save on memory in classifying flows to these applications. For example, BLINC considers only the first 50 non-payload flows from a host during an interval to detect an address scan from that host. Thereafter, it ignores all non-payload flows from that host during the current interval. Our templates instead consider each flow for sampling, without mimicing these optimizations. Nonetheless, the sampling ratio obtained for these 2 applications as well is better than 10. Due to the absence of any information loss in constructing templates that are equivalent to BLINC's graphlets, there is no need to infer filters for better relevance of sampled records.

These results demonstrate the expressive power of our template language. We were able to tailor the sampling to capture the variance in requirements across the range of Snort/Bro rulesets, and BLINC graphlets. Our results also point towards long-term prevalence of the local traffic characteristics that are represented by the filters we inferred for both Snort and Bro alerts. Thus, the lack of information in flow records can be compensated by having a learning phase precede the deployment of the sampler into operation.

Our intent in sampling flow records of interest to the above applications is not to modify them to take sampled records as input. Nor do we seek to propose intrusion detection to be done just on the sampled flow records. We believe the hints about traffic characteristics gathered from the sampled records can be used to *tailor* the execution of each application. Flow records can be gathered only after sampling at many network locations today, and such records are being used to glean information, e.g., about attacks. Our results show that the sampling can be tailored for such needs while still tolerating sampling constraints.

3.5 Handling multiple input templates

So far, we have considered the sampling of flow records w.r.t each input template in isolation. However, in practice, several templates will be fed in as input simultaneously to the sampling module. As a result, even though a sufficiently low sampling probability is obtained for each input template, the probability of a flow record being sampled may turn out to be high in aggregate. For example, in the case of Snort, there will be one input template corresponding to each Snort rule. Consequently, for the data shown in Figure 2(a), even though sampling better than 1-in-100 is obtained for most rules when sampling using filters, the aggregate sampling probability is 0.5. Such high sampling probabilities may not be affordable in high traffic environments where resources are scarce.

To address this problem, some of the flow records will need to be probabilistically dropped. The flow collector utilizes memory/CPU to match records with templates, and disk space/bandwidth to write the sampled records to disk. Dropping records prior to matching them with templates will save on both memory/CPU and disk space/bandwidth. However, this would adversely impact the matching of records to templates, and the records eventually sampled may not be of much use to the application. Therefore, we consider probabilistically dropping records after they have been matched with templates but before writing them to disk.

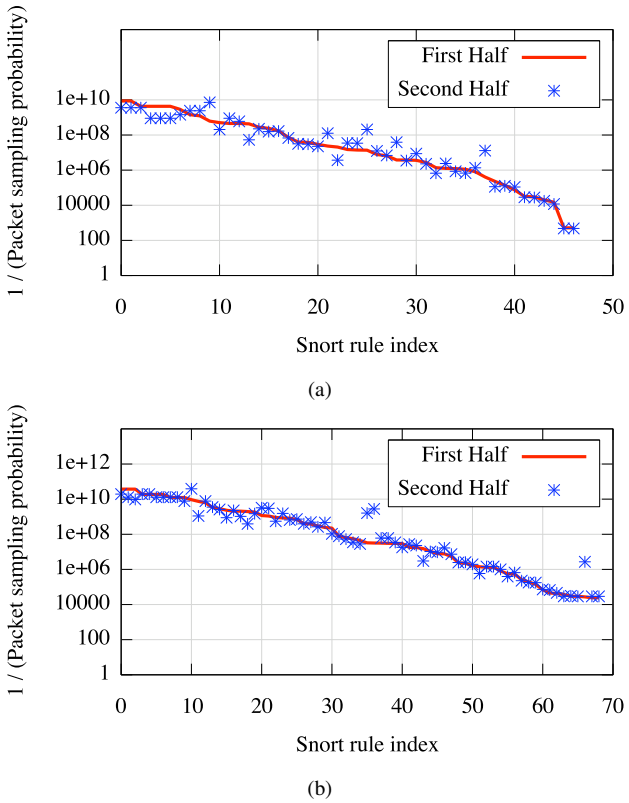


Figure 5: Comparison of optimal sampling probabilities on the first and second halves of (a) *Corp* and (b) *Univ* with Snort's templates as input. Note log-scale on y axis.

Our key insight is that some templates are more lax than others, and records matching such templates should be dropped with higher probability. The administrator at the site where the flow records are being gathered can determine the laxness of each template by monitoring the number of records that match the template during the learning phase. Depending on the resources available, he can then appropriately associate each template with a probability; records matching the template will be chosen with this probability to be written to disk. For example, if $\frac{1}{10}^{th}$ of the records are seen to match a particular template, but the flow collector has sufficient resources only to perform 1-in-100 sampling, this template will be associated with a probability of 0.1.

However, every template can be associated with a probability as above only if the fraction of flow records that match the template remains consistent over time. To evaluate whether this hypothesis holds in the context of Snort, we compared the optimal sampling probability for every Snort rule across either half of both datasets. Figure 5(a) shows that for most of the Snort rules, the optimal sampling probability remains at the same order of magnitude in either half of *Corp*. The same is seen to hold in *Univ* as well, as shown in Figure 5(b). Further, in the case of Bro, Table 2 shows that in both datasets, the optimal probability for sampling flows corresponding to sensitive connections remains within an order of magnitude across halves. We do not consider the same evaluation for BLINC since, unlike intrusion detection where the objective is to identify the subset of traffic that is anomalous, traffic classification attempts to classify all input traffic.

Note that the dropping of flow records after they have been matched with templates does affect the conclusions drawn from the sam-

pled flow records. This is inevitable given that resources are limited. If a template is too lax, it is infeasible to gather all flow records that match the template. The approach described above of dropping records based on template-specific probabilities strikes the right balance in the tradeoff between accuracy and resource consumption—records corresponding to templates that are lax are dropped selectively to save on resources without comprising the accuracy of sampling for other templates.

4. RELATED WORK

Prior work on flow record sampling falls into three categories—languages for specification of sampling that is more flexible than 1-in-n, sampling algorithms to be used in routers and flow collectors for gathering flow records, and algorithms for deriving estimates of various properties from sampled flow records. A detailed examination of the literature on passive measurement related sampling can be found in [7].

The packet sampling working group PSAMP [8] at the IETF has been working on a protocol specification, as well as sampling and filtering techniques. Primarily sampling focuses on position of the packet, content of the packet, and a probabilistic factor. Schemes for sampling envisioned include a deterministic function that has triggers that are based on spatial position of packets or time. Although considerable work has occurred in the ambit of PSAMP, there is no provision for accepting hints from application to tailor the sampling. Our work is complementary to the PSAMP effort.

Since we investigate viability of application-specific sampling for two applications, intrusion detection and traffic classification, we also look at related work in these arenas.

Flexible Sampling: Instead of performing 1-in-n sampling with the same n for all traffic, Cisco's Netflow input filters [2] enable segregation of traffic into classes and specification of varying n across classes. In addition, Flexible Netflow [1] lets the router administrator select the set of fields that are kept track of in generating flow summaries. Compound and stateful signatures in Juniper's Netscreen-IDP suite [16] are similar in spirit to our template language. However, their language specification is not open, and hence, their extensibility is not apparent, e.g., whether they can represent the needs of time-triggered applications.

ProgME [32] introduces *flowsets*, a way to group flows based on specific application needs. They have a language for composing predicates that allows for adaptation to traffic changes. If netflow data is gathered using their proposed techniques (different from how it is being carried out today) they will be able to provide a flexible environment for multiple applications. However, the language used by ProgME only captures unions, intersections, and negations of flowsets, but not a sequence of flows as required by record-triggered applications.

Gigascope [4] is a data stream management system specialized for network traffic analysis used with high performance packet capture hardware. Gigascope queries are specified in a SQL-like language with some extensions, but with restrictions such as the requirement that aggregate and join operations be run over a fixed size time window. Aurora [6] is a DBMS that allows data to be streamed through its collection of a small number of primitive operations (such as windowed operators, an operator to partition streams into separate windows, etc.). A variety of query types are supported and the system has query optimization capabilities. Both Gigascope and Aurora are limited by their SQL-like languages in their ability to capture application requirements.

Sampling Techniques: Flow Slices [19] permits independent tuning of packet sampling, flow sampling and flow record reporting to control the CPU, memory and bandwidth usage in a router. Rather than performing packet sampling and flow sampling independently, the *sample and hold* algorithm [15] determines which flows are large and samples all packets on those. Duffield *et al.* [11] solved the problem of sampling a fixed number of records in each interval while still being able to derive unbiased estimates of volume. The system described by Keys et al. [18] proposes adaptable summaries of traffic with a control on resource usage that is achieved by graceful degradation of accuracy. Their focus is on detecting heavy hitters.

The common theme across these various sampling techniques is their fairly narrow focus of enabling either estimation of traffic volumes or detection of causes of high traffic volumes. Many large network entities have to monitor the amount of traffic they exchange with each other, and sampling gives them a cheaper way to do this. However, several other applications that use flow records have recently emerged. Examples include identification of P2P traffic characteristics [20], and analysis of traffic anomalies [3, 22]. Also, applications that operate on packet streams can benefit from information in flow records in high traffic volume environments. For most of these applications, flow records can provide useful information only if the records for flows of interest to the application are preferentially sampled.

Algorithms for Deriving Estimates: The problem of deriving useful estimates from sampled flow records has also been extensively studied. Duffield *et al.* addressed the problems of estimating billing information [9] and flow distributions [10] from sampled flow records. Threshold sampling [12] optimally trades-off the expected number of flows sampled against the variance of volume estimates. The optimal methodology for merging sampled measurements [13] has also been determined. Like sampling techniques, most of this work is restricted to determining estimates of traffic volumes.

Intrusion Detection: Flow records have been used previously for detecting anomalous or malicious traffic. LADS [30] triggers gathering of flow records based on SNMP data, and then uses these gathered flow records to detect Distributed Denial of Service (DDoS) attacks. Flow records in the sFlow format have been used for continuous network-wide monitoring of network traffic [26]. We investigate whether we can selectively sample flow records of relevance to Snort [28] and Bro [25]. First, we determine rules that map these applications' interest to flow records. Second, we inspect traffic at two different locations to identify local filters that help decrease the sampling probability. Automated filter detection has been employed previously in [29] and [14].

Recently, there has been research examining the impact of sampling on anomaly detection. First, examination of unsampled flows against a packet trace captured during a worm outbreak [5] showed that flow counts were significantly affected due to sampling; if single packet flows are missed at high sampling rates, anomalies would not be caught. However, entropy-based summarizations have the ability to retain information related to worm-like attacks. If information about the nature of attacks of interest can be supplied to the sampler, the sampled data stream would naturally be more suitable for detection. Second, evaluation [23] of four different sampling techniques against a backbone trace showed expected but differing bias across the techniques; the authors advocate focusing on alternate methods that can reduce information loss. We believe sampling using the language we propose could be one such method.

Traffic classification: Lately there has been considerable interest in automatic traffic classification. Beyond BLINC, there have been approaches ranging from statistical approaches [27, 24] to machine learning techniques [31, 33]. BLINC has focused on the transport layer to help with classification without requiring packet

payload or knowledge of port numbers. Recently there has been work in separating elephants (persistent flows with large number of packets) from mice (short lived flows with few packets) using information theory [21], without relying on prior flow distribution knowledge and low packet processing overhead.

5. CONCLUSIONS AND FUTURE WORK

We introduced an application-agnostic language that can be used to tailor the sampling of flow records such that records of relevance to applications are selected. We demonstrated the viability of our approach by considering three popular applications—BLINC, Snort, and Bro— as examples. In all three cases, our template language was expressive enough for us to characterize the application's traffic of interest. We were able to choose flow records of relevance to each application by folding in local traffic information into the corresponding template.

One of the avenues for future work is building an efficient sampler that leverages the expressive power of our template language. The focus in developing a sampler needs to be on optimizing the sampler's resource utilization. Towards this goal, two issues will need to be addressed. First, the sampler needs to be moved from the flow collector to the router, which necessitates the incorporation of packet sampling into our framework. Second, the sampler's state when given several templates as input needs to be optimized.

Acknowledgments

Steven Gao, George Lazarou, Chris Olsen, Oliver Spatscheck, and Gang Yao at AT&T, David Ruddick and David Sinn at the University of Washington helped us with the infrastructure for gathering data—our thanks to them. We thank Dave Kormann for helpful discussions in diagnosing the problems in sniffing traffic for an extended period, Ramana Kompella, Arvind Krishnamurthy, and Vyas Sekar for comments on earlier drafts of this work, and Glenn Fowler for help with *dss*. We thank the anonymous reviewers for their detailed reviews and the CCR editor for his help during the review process.

6. REFERENCES

[1] Flexible Netflow. `http://www.cisco.com/en/US/products/ps6965/products_ios_protocol_option_home.html`.

[2] Netflow input filters. `http://www.cisco.com/en/US/products/sw/iosswrel/ps5207/products_feature_guide09186a00801d3108.html`.

[3] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *IMW*, 2002.

[4] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *IMC*, 2003.

[5] D. Brauckhoff et al. Impact of traffic sampling on anomaly detection metrics. In *IMC*, 2006.

[6] D. Carney et al. Monitoring streams—a new class of data management applications. In *VLDB*, 2002.

[7] N. Duffield. Sampling for passive Internet measurement: A review. *Statistical Science*, 19(3):472–498, 2004.

[8] N. Duffield. A framework for packet selection and reporting, 2007. IETF draft: psamp-framework-11.

[9] N. Duffield, C. Lund, and M. Thorup. Charging from sampled network usage. In *IMW*, 2001.

[10] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *SIGCOMM*, 2003.

[11] N. Duffield, C. Lund, and M. Thorup. Flow sampling under hard resource constraints. In *SIGMETRICS*, 2004.

[12] N. Duffield, C. Lund, and M. Thorup. Learn more, sample less: Control of volume and variance in network measurement. *IEEE Transactions on Information Theory*, 51:1756–1775, 2005.

[13] N. Duffield, C. Lund, and M. Thorup. Optimal combination of sampled network measurements. In *IMC*, 2005.

[14] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM*, 2003.

[15] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM TOCS*, 2003.

[16] Juniper Networks. Using compound signatures to protect against complex attacks, 2004.

[17] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel traffic classification in the dark. In *SIGCOMM*, 2005.

[18] K. Keys, D. Moore, and C. Estan. A robust system for accurate real-time summaries of Internet traffic. In *SIGMETRICS*, 2005.

[19] R. R. Kompella and C. Estan. The power of slicing in Internet flow measurement. In *IMC*, 2005.

[20] B. Krishnamurthy and J. Wang. Traffic classification for application specific peering. In *IMW*, 2002.

[21] S. Kundu, S. Pal, K. Basu, and S. Das. Fast classification and estimation of Internet traffic flows. In *PAM*, 2007.

[22] A. Lakhina, M. Crovella, and C. Diot. Characterization of network-wide anomalies in traffic flows. In *IMC*, 2004.

[23] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is sampled data sufficient for anomaly detection? In *IMC*, 2006.

[24] A. W. Moore and D. Zuev. Traffic classification using bayesian analysis techniques. In *SIGMETRICS*, 2005.

[25] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[26] J. Reves and S. Panchen. Traffic monitoring with packet-based sampling for defense against security threats. *InMon Technology Whitepaper*, 2002.

[27] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification. In *IMC*, 2004.

[28] Snort. `http://www.snort.org`.

[29] C. Taylor and J. Alves-Foss. NATE: Network analysis of anomalous traffic events, a low-cost approach. In *New Security Paradigms Workshop*, 2001.

[30] V. Sekar et al. LADS: Large-scale automated DDoS detection system. In *USENIX Annual Technical Conference*, 2006.

[31] N. Williams, S. Zander, and G. Armitage. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *ACM CCR*, 2006.

[32] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards programmable network measurement. In *SIGCOMM*, 2007.

[33] S. Zander, T. Nguyen, and G. Armitage. Automatic traffic classification and application identification using machine learning. In *IEEE LCN*, 2005.