Rule-Based Anomaly Detection on IP Flows

Nick Duffield*, Patrick Haffner*, Balachander Krishnamurthy*, Haakon Ringberg‡
*AT&T Labs—Research, Florham Park, NJ 07932
‡Dept. of Comp. Science, Princeton University, Princeton, NJ 08544
{duffield,haffner,bala}@research.att.com, haakon@cs.princeton.edu

***Abstract*—Rule-based packet classification is a powerful method for identifying traffic anomalies, with network security as a key application area. While popular systems like Snort are used in many network locations, comprehensive deployment across Tier-1 service provider networks is costly due to the need for high-speed monitors at many network ingress points. Since ISPs already collect flow statistics ubiquitously, can we use it for detecting the same anomalies as the packet based rules in spite of aggregation and absence of payload information? We exploit correlations between packet and flow level information via a machine learning (ML) approach to associate packet level alarms with a feature vector derived from flow records on the same traffic. We describe a system architecture for network-wide flow-alarming and describe the steps required to establish a proof-of-concept. We evaluate prediction accuracy of candidate ML algorithms on actual packet traces. The duration of prediction effectiveness is an issue for ML approaches and more so in resource intensive network applications. Initial results show little impairment of performance over periods of one or two weeks.**

I. Introduction

A. Motivation

Detecting unwanted traffic is a crucial task in managing data communications networks. Detecting network attack traffic, and non-attack traffic that violates network policy, are two key applications. Many types of unwanted traffic can be identified by rules that match known signatures. Rules may match on a packet's header, payload, or both. The 2003 Slammer Worm [1], which exploited a buffer overflow vulnerability in the Microsoft SQL server, was matchable to a signature comprising both packet header fields and payload patterns.

Packet inspection can be carried out directly in routers, or in ancillary devices observing network traffic, e.g., on an interface attached to the network through a passive optical splitter. Special purpose devices of this type are available from vendors, often equipped with proprietary software and rules. Alternative software systems such as Snort [2] can run on a general purpose computer, with a language for specifying rules created by the user or borrowed from a community source.

In any of the above models, a major challenge for comprehensive deployment over a large network, such as a Tier-1 ISP, is the combination of network scale and high capacity network links. Packet inspection at the network edge involves deploying monitoring capability at a large number of network interfaces (access speeds from OC-3 to OC-48 are common). Monitoring in the network core is challenging since traffic is concentrated through higher speed interfaces (OC-768 links are increasingly being deployed). Wherever the traffic is monitored, many hundreds of rules may need to be operated concurrently. Whereas fixed-offset matching is cheap computationally and has known costs, execution of more complex queries may hit computational bandwidth constraints. Even when inspection is operated as a router feature, there may be large licensing costs associated with its widespread deployment.

B. Signature-based Detection on IP Flows

An intrusion detection system that could inspect every network packet would be ideal, but is impractical. Signature-based detection systems such as Snort have been widely deployed by enterprises for network security, but are limited by the scaling factors described above. We have therefore developed an architecture that can translate many existing packet signatures to instead operate effectively on IP flows. Flow statistics are compact and collected ubiquitously within most ISPs' networks, often in the form of NetFlow [3].

Our work does not supplant signature-based detection systems, but rather extends their usefulness into new environments where packet inspection is either infeasible or undesirable. We wish to construct rules at the flow level that accurately reproduce the action of packet-level rules. In other words, an alarm should ideally be raised for flows that are derived from packets that would trigger packet-level rules. Our methods are probabilistic in that the flow level rules do not reproduce packet level rules with complete accuracy; this is the price we pay to be scalable.

The idea of deriving flow-level rules from the header portion of a packet-level rule has been proposed in [4], but this technique only applies to rules that exclusively inspect a packet's header. What can be done for rules that contain predicates that match on a packet's payload? Ignoring the rule or removing the predicates are both unsatisfactory options, as they will lead to heavily degraded detection performance in general. Signatures that inspect a packet's payload can still be effectively learned if there is a strong association between features of the flow header produced by this packet and the packet's payload. For example, the Slammer Worm infects new host computers by exploiting a buffer overflow bug in Microsoft's SQL server; these attack packets contain known payload signatures in addition to targeting a specific UDP port on the victim host. The Snort signature to detect these packets utilizes both these pieces of information to improve detection. An exhaustive system for translating packet rules into flow rules must leverage these correlations between the packet payload and flow header in order to mitigate the impact of losing payload information.

Some signatures exhibit a strong association between payload and flow-header information even though no correlation is implied in the original packet signature. This can occur either because the human author of the signature was unaware of or disregarded this piece of information (*e.g.*, the unwanted traffic very frequently uses a particular destination port, even though this was not specified in the packet signature), or because the association exists between the payload and flow-header features that have no packet-header counterpart (*e.g.*, flow duration). For this reason, our architecture leverages Machine Learning (ML) algorithms in order to discover the flow-level classifier that most successfully approximates a packet signature. The essential advantage of ML algorithms is their ability to learn to characterize flows according to predicates that were not included in the original packet-level signature.

C. Contribution

The primary contribution of this paper is the presentation of an ML-based architecture that can detect unwanted traffic using flow signatures. These flow signatures are learned from a reference set of packet signatures and joint packet/flow data. We evaluate our system on traces from and signatures used by a medium-sized enterprise. Our results show that ML algorithms can effectively learn many packet signatures including some that inspect the packet payload. We also demonstrate that our system is computationally feasible in that it: (1) can relearn the packet signatures long before this becomes necessary due to inherent data drift, and (2) the learned classifiers can operate at very high speeds. This is demonstrated both analytically and empirically.

We analyze our results with an emphasis on understanding why some signatures can be effectively learned whereas others cannot. To this end, we also present a taxonomy of packet signatures that *a priori* separates them into sets (A) that our system will be able to learn perfectly, (B) that our system is likely to learn very well, or (C) where the accuracy of our learned classifier varies based on the nature of the signature. For signatures that fall into classes (B) or (C), where there is *a priori* uncertainty regarding how well our system will perform, we detail the properties of the signatures that are successfully learned using examples from our set of signatures.

The rest of our paper is organized as follows. We discuss related work in Section II. A taxonomy of packet signatures is presented in Section III. In Section IV we discuss the relevant aspects of how signature-based detection systems are used in practice, including some specifics on Snort rules and of flow level features that we employ. The operation of ML algorithms, and an algorithm that we find effective, namely, Adaboost, are reviewed in Section V. Section VI describes our dataset and experiment setup; our performance evaluation methodology is presented in Section VII, including detection accuracy metrics we have used. This prepares for our experimental evaluation results in Section VIII, in addition to further analysis of the signatures whose detection performance our *a priori* taxonomy cannot predict. A proposal for how our system could fit into a distributed anomaly detection architecture is then presented in Section IX The computational efficiency of our system, both in terms of learning and classifying flows according to given packet-level signatures, is discussed in Section X. before we present our conclusions in Section XI.

II. Related Work

There is an extensive recent literature on automating the detection of unwanted traffic in communications networks, most importantly, detection of email spam, denial of service attacks and other network intrusions. Anomaly detection has been used to flag deviations from baseline behavior of network traffic learned through various unsupervised methods, including clustering, Bayesian networks, PCA analysis and spectral methods; see, e.g., [5], [6], [7], [8], [9], [10]. Our approach is different to these: rather than alarming unknown unusual events based on deviation from observed norms, we regard the set of events alerted by packet rules as representing the most complete available knowledge. The function of ML is to determine how best to reproduce the alerts at the flow level.

ML techniques have been used for traffic application classification. Approaches include unsupervised learning of application classes via clustering of flow features and derivation of heuristics for packet-based identification [11]; semi-supervised learning from marked flow data [12] and supervised learning from flow features [13], [14].

III. A Packet Signature Taxonomy

We adopt the following model and classification for packet rules. A packet rule is specified by a set of predicates that are combined through logical AND and OR operations. We classify three types of predicate: flow-header (FH), packet payload (PP), and meta-information (MI) predicates.

FH predicates involve only packet fields that are reported exactly in any flow record consistent with the packet key. This include source and destination IP addresses and UDP/TCP ports, but exclude packet header fields such as IP identification (not reported in a flow record) and packet length (only reported exactly in single packet flows).

PP predicates involve the packet payload, i.e. excluding network and transport layer headers present.

MI predicates involve only packet header information that is reported either inexactly or not at all in the flow record (e.g., the IP ID field). From the above discussion, packet length is MI, as are TCP flags, because being cumulative over flows of packets, they are reported exactly only for single-packet flows.

Packet rules may contain multiple predicates, each of which may have different types of (FH, PP, MI) associated with it. We give a single type to the rule itself based on the types of predicates from which it is composed. We partition the set of possible packet rules into disjoint classes based on the types of predicates present. The classification sits well with the performance of our ML-method, in the sense that rule class is a qualitative predictor of accuracy of learned flow-level classifiers. Our packet rule classification is as follows

Header-Only Rules: comprise *only* FH predicates.

Payload-Dependent Rules: include at least one PP predicate.

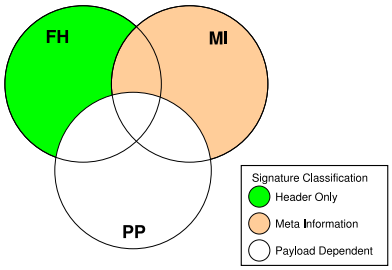

Fig. 1. Packet Rule Classification: FH (flow header), PP (packet payload), MI (meta-information) indicate rule attributes according to predicate classes; disjoint packet rule classification illustrated by different colors.

Meta-Information Rules: include no PP predicates, do include MI predicates, and may include FH predicates.

The relationship between the classification of packet rules and the classification of the underlying predicates is illustrated in Figure 1; each circle illustrates the set of rules with attributes corresponding to the predicate classification FH, PP, and MI. The packet rule classification is indicated by colors.

IV. Packet and Flow Rules in Practice

Snort [2] is an open-source IDS that monitors networks by matching each packet it observes against a set of rules. Snort can perform real-time traffic and protocol analysis to help detect various attacks and alert users in real time. Snort employs a pattern matching model for detecting network attack packets using identifiers such as IP addresses, TCP/UDP port numbers, ICMP type/code, and strings obtained in the packet payload. Snort's rules are classified into priority classes, based on a global notion of the potential impact of alerts that match each rule. Each Snort rule is documented along with the potential for false positives and negatives, together with corrective measures to be taken when an alert is raised. The simplicity of Snort's rules has made it a popular IDS. Users contribute rules when new types of anomalous or malicious traffic are observed. 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.

Our flow-level rules were constructed from the following features of flow records: source port, destination port, #packets, #bytes, duration, mean packet size, mean packet inter-arrival time, TCP flags, protocol, ToS, "source IP address is part of Snort home net", "destination IP address is part of Snort home net", "source IP address is an AIM server", "destination IP address is an AIM server". The Snort home net is commonly configured to whatever local domain the operator desires to protect from attacks originating externally.

We construct *flow level predicates* in the following ways:

(1) For categorical features like protocol or TCP flags, we use as many binary predicates as there are categories. For example, if the protocol feature could only take on the values {ICMP, UDP, TCP} then an ICMP packet would be encoded as the predicate ICMP=1, UDP=0, and TCP=0.

(2) For numerical features such as #packets, we want to be able to finely threshold them, so that a rule with a predicate specifying, e.g. an exact number of packets, can be properly captured. Our predicates take the form "feature $>$ threshold".

Our system seeks to leverage ML algorithms in order to raise Snort alerts on flow records. To train our ML algorithms we require concurrent flow and packet traces so that the alerts that Snort raises on packets can be associated with the corresponding flow record. "Correspondence" here means that the packets and flow originate from the same underlying connection. In other words, if Snort has raised an alert on a packet at time t then we locate the flow with the same IP 5-tuple, start time t_s, and end time t_e such that $t_s \leq t \leq t_e$. We then associate the packet alert with the flow. A single packet may raise multiple Snort alerts, and a single flow will often correspond to a sequence of packets, which means that individual flows can be associated with many Snort alerts.

V. Machine Learning Algorithms

Formally our task is as follows. For each Snort rule our training data takes the form of a pair (x_i, y_i) where flow i has flow features x_i, and $y_i \in \{-1, 1\}$ indicates whether flow i triggered this Snort rule. Our aim is to attribute to each Snort rule a score in the form of a weighted sum $\sum_k w_k p_k(x)$ over the flow level predicates $p_k(x)$ described in Section IV. When this score exceeds an *operating threshold* θ, we have an *ML Alarm*. Since ML alarms should closely mimic the original Snort alarms y_i, the weights w_k are chosen to minimize the classification error $\sum_i I(y_i \neq \mathsf{sign}(\sum_k w_k p_k(x) - \theta)$. However, deployment considerations will determine the best operating threshold for a given *operating point*.

Supervised linear classifiers such as Support Vector Machines (SVMs) [15], Adaboost [16] and Maximum Entropy [17] have been successfully applied to many such problems. There are two primary reasons for this. First, the convex optimization problem is guaranteed to converge and optimization algorithms based either on coordinate or gradient descent can learn millions of examples in minutes (down from weeks ten years ago). Second, these algorithms are regularized and seldom overfit the training data. This is what our fully automated training process requires: scalable algorithms that are guaranteed to converge with predictable performance.

Preliminary experiments established that, on average, Adaboost accuracy was significantly better than SVMs. In the remainder of this section we therefore highlight the properties of Adaboost that make it well-suited for our application. A linear algorithm like Adaboost works well here because the actual number of features is large. In theory, each numerical feature (*e.g.*, source port or duration) may generate as many flow level predicates of the form "feature $>$ threshold" (such predicates are called *stump classifiers*) as there are training examples. In practice, this potentially large set of predicates does not need to be explicitly represented. Adaboost has an incremental greedy training procedure that only adds predicates needed for finer discrimination [16].

Flow type	wk 1	wk 2	wk 3	wk 4
Neg:no alerts	202.9	221.8	235.9	251.6
Unique neg.	41.8	48.3	42.7	48.7
Pos:some alert	6.7	7.2	6.5	6.9
Unique pos	0.1	0.1	0.1	0.1

TABLE I
NUMBER OF FLOWS IN MILLIONS PER WEEK

Protocol	Flag value	Alerts	No alert
ICMP	1	.383	88.5
TCP	6	.348	55.3
UDP	17	6.79	77.1

TABLE II
NUMBER OF FLOWS IN MILLIONS PER PROTOCOL FOR WEEK 2

Good generalization is achieved from classifiers that represent the "simplest" linear combination of flow-level predicates. Adaboost uses an L_1 measure of simplicity that encourages sparsity, a property that is well matched to our aim of finding a small number of predicates that are closely related to the packet level rules. This contrasts with the more relaxed L_2 measure used by SVM's, which typically produces more complex classifiers. Finally, while Adaboost is known for poor behavior on noisy data, the low level of noise in our data makes the learning conditions ideal. In preliminary experiments, we observe a similar behavior with L_1-regularized Maximum Entropy[17], an algorithm that is much more robust to noise.

VI. Data Description and Evaluation Setup

The data was gathered at a gateway serving hundreds of users during August–September 2005. We examined all traffic traversing an OC-3 link attached to a border router, gathered via an optical splitter. A standard Linux box performed the role of a monitor reading packets via a DAG card. Simultaneously, unsampled netflow records were also collected from the router. Snort rules in place at the site were used for the evaluation. The traffic represented 5 Terabytes distributed over 1 Billion flows over 29 days, i.e., an average rate of about 2MBytes/second. The average number of packets per flow was 14.5, and 55% of flows comprised 1 packet.

We split the data into 4 weeks. Week 1 is used for training only, week 2 for both training and testing and weeks 3-4 for testing only. Table I reports the number of flows each week. The 200–250 million examples collected each week would represent a major challenge to current training algorithms. Fortunately, the number of unique examples is usually 40–50 million per week, and of these only about 100,000 contain an alert. These can train optimized implementations of Adaboost or SVMs in a span of hours. Removing purely deterministic features greatly simplifies the training problem by reducing the number of examples; it also slightly improves performance. The two main deterministic features are:

source IP is part of local network: Snort rules usually assume that alerts can only be caused by external flows, which means that they require this feature to be 0. After computing unique flow statistics, there were 54 million local and 167 million external flows which are not alerts, zero local and 7 million external flows which are alerts. Making a boolean decision that all local flows are safe, prior to running the classifier, reduces the training data by 54 million examples.

protocol: Snort rules only apply to a single protocol, so splitting the flows into ICMP, TCP and UDP defines 3 smaller learning problems, minimizing confusion. Table II shows how week 2 can be split into 3 subproblems, where the most complex one (UDP) only has 6.79 million alert flows and 77.1 million no-alert flows.

Alerts of 75 different rules were triggered over the 4 week trace. We retained the 21 rules with the largest number of flows over weeks 1 and 2; the resulting rules are listed in Table III. The second column reports the total number of flows associated with the rule over week 1 and 2, which range from 13 million to 1360 (note that most rules are evenly distributed over the 4 weeks). The third column reports the number of unique flows, which is representative of the complexity of a rule, being the number of positive examples used in training. (The remaining columns are discussed in Section VIII).

VII. Detection Performance Criteria

Each rule is associated with a binary classifier that outputs the confidence with which the rule is detected on a given flow. A detection is a boolean action, however, and therefore requires that an operating threshold is associated with each classifier. Whenever the classifier outputs a confidence above the operating threshold, an alarm is raised. It is customary in the machine learning literature to choose the operating threshold that minimizes the classification error, but this is not necessarily appropriate in our setting. For example, a network operator may choose to accept a higher overall classification error in order to minimize the *False Negative* rate. More generally, the network operators is best equipped to determine the appropriate trade-off between the *False Positive* (FP) and *True Positive* TP rates. The **Receiver Operating Characteristics** (ROC) curve presents the full trade-off for binary classification problems by plotting the TP rate as a function of the FP rate. Each point on the ROC curve is the FP and TP values for a specific confidence (*i.e.*, operating threshold) between 0 and 1.

The ROC curve is useful for network operators because it provides the full trade-off between the FP and TP rates, but this also makes it a poor metric for us when evaluating a number of rules in a number of different settings. For our purposes we require a single threshold-independent number that must account for a range of thresholds. The most studied such measure is the **Area Under the ROC Curve** (AUC), but all our experiments return AUC values better than 0.9999. Besides the fact that such values make comparisons problematic, they are often meaningless. The **Average Precision** (AP), defined in (1) below, provides a pessimistic counterpart to the optimistic AUC. When setting the threshold at the value of positive example x_k, the numbers of total and false positives are $TP_k = \sum_{i=1}^{n_+} 1_{x_k \leq x_i}$ and and $FP_k = \sum_{j=1}^{n_-} 1_{x_k \leq z_j}$, where i and j label the n_+ positive examples and n_- negative

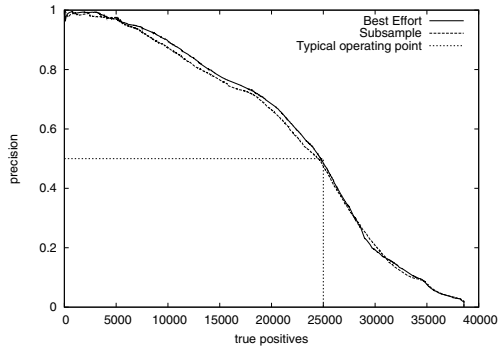

Fig. 2. Precision vs. number of true positives for the EXPLOIT ISAKMP rule training on week1 and testing on week2

examples z_j respectively. The precision at threshold x_k is the fraction of correctly detected examples $\frac{TP_k}{TP_k+FP_k}$ and we compute its average over all positive examples

$$AP = \frac{1}{n_+} \sum_{k=1}^{n_+} \frac{TP_k}{TP_k + FP_k} \quad (1)$$

The AP reflects the negative examples which score above the positive examples, and, unlike the AUC, ignores the vast majority of negative examples whose scores are very low. A benefit of the AP metric is that it is more interpretable. Suppose we run Snort until it detects a single alert, and then set up the detection threshold at the classifier output for this alert. Assuming the alerts are I.I.D., an AP of p means that, for each true positive, one can expect $\frac{1-p}{p}$ false negatives.

Let us illustrate what AP means with an example drawn from the results detailed in the next section. Figure 2 plots the precision as a function of the number of TP for the EXPLOIT ISAKMP rule: the AP corresponds to the area under this curve. We can see what a comparatively low AP of 0.58 for this rule means in terms of the operating curve. We are able to alert on say 25,000 of the Snort events (about 2/3), while suffering the same number of false negatives, *i.e.* a precision of 0.5. We will see in the next section that for many other rules we do far better, with AP close to 1, leading to very small false positive rates. Moreover, we will explain how a classifier with an AP of 0.5 can still be very useful to a network operator.

VIII. Experimental Results

A. Baseline Behavior

The average precisions of our learned flow detectors are reported in Table III. We have grouped alerts according to the taxonomy presented in Section III. For each category we perform a simple *macro-average*, where the AP for each rule is given equal weight, which is reported in the *average* row beneath each rule group. The *baseline* column in Table III reports the AP from training on one full week of data and testing on the subsequent week. We perform two such experiments: the *wk1-2* column uses week 1 for training and week 2 for testing whereas *wk2-3* uses week 2 for training and week 3 for testing. For header and meta-information rules, the baseline results give an AP of at least 0.99 in all cases. Payload rules exhibit greater variability, ranging from about 0.4 up to over 0.99. Our analysis will illuminate the different properties of rules that lead to this variation in ML performance.

There were two payload rules that exhibited dramatically lower AP than the others; these are listed at the end of Table III and not included in the macro-average. A detailed examination of the underlying Snort rules showed these to be relatively complex and designed to alarm on a mixed variety of predicates. We posit that the complexity of the Snort rules contributes to the difficulty in accurately characterizing them based on flow features.

B. Data Drift

The main information provided in Table III concerns the dependence of the AP as a function of the temporal separation between the training data and the test data. Measuring how performance drifts over time is critical, as it determines how often retraining should be applied. While our baseline corresponds to a 1-week drift, *wk1-3* indicates a 2 week drift: it can either be compared to *wk1-2* (same training data, drifted test data) or *wk2-3* (drifted training data, same test data). In both cases, the difference from a 1-week drift to a 2-week drift is often lower than the difference between *wk1-2* and *wk2-3*: this suggests that the impact of a 2-week drift is too low to be measurable. On the other hand, the loss in performance after a 3 week drift (*wk1-4*) is often significant, in particular in the case of Payload and Meta-Information rules.

C. Sampling of Negative Examples

Because the number of negative examples far exceeds the number of positive training examples, (*i.e.*, the vast majority of packets—and flows—do not raise any Snort alarms), we expect that sampling to reduce the number of negative examples will have minimal impact on detection accuracy, but will drastically reduce the training time. We wish to preferentially sample examples whose features are more common, or conversely, avoid the impact of noise from infrequently manifest features. For this reason we group the negative examples into sets with identical features, then apply Threshold Sampling [18] to each group as a whole. This involves selecting the group comprising c examples with probability $\min\{1, c/z\}$ where z is chosen so as to sample a target proportion of the examples.

The results for a sampling rate of 1 in 100 negative examples are shown in the two columns labeled *Sampling*, rightmost in Table III. When comparing either the *wk1-2* or the *wk2-3* columns in the baseline and in the sampled case, there is a measurable loss in performance. This loss is small relative to fluctuations in performance from one week to another, however, which suggests that sampling negative training examples is an effective technique. In this example, sampling speeds up training by about a factor of 6. Without sampling, training a single rule takes on average 1 hour on a single Xeon 3.4GHz processor, but can be reduced to 10 minutes with sampling.

Alert message	Number of flows over weeks 1-2		Average Precision for wkA-B (week A=train, B=test)					
			Baseline		Drift		Sampling	
	total	unique	wk1-2	wk2-3	wk1-3	wk1-4	wk1-2	wk2-3
Header								
ICMP Dest. Unreachable Comm. Administratively Prohib.	154570	12616	1.00	1.00	1.00	1.00	1.00	1.00
ICMP Destination Unreachable Communication with Destination Host is Administratively Prohibited	9404	3136	0.99	0.99	0.98	0.99	0.99	0.98
ICMP Source Quench	1367	496	1.00	1.00	1.00	1.00	1.00	1.00
average			1.00	0.99	0.99	0.99	1.00	0.99
Meta-information								
ICMP webtrends scanner	1746	5	1.00	0.99	0.99	0.99	0.90	0.99
BAD-TRAFFIC data in TCP SYN packet	2185	2145	1.00	1.00	1.00	0.99	1.00	1.00
ICMP Large ICMP Packet	24838	1428	1.00	1.00	1.00	1.00	1.00	1.00
ICMP PING NMAP	197862	794	1.00	1.00	1.00	1.00	0.61	1.00
SCAN FIN	9169	7155	0.99	1.00	1.00	0.86	0.99	1.00
(spp stream4) STEALTH ACTIVITY (FIN scan) detection	9183	7169	1.00	1.00	1.00	0.87	1.00	1.00
average			1.00	1.00	1.00	0.95	0.92	1.00
Payload								
MS-SQL version overflow attempt	13M	28809	1.00	1.00	1.00	1.00	1.00	1.00
CHAT AIM receive message	1581	1581	0.66	0.57	0.60	0.65	0.56	0.30
EXPLOIT ISAKMP 1st payload length overflow attempt	76155	65181	0.59	0.58	0.57	0.57	0.58	0.56
ICMP PING CyberKit 2.2 Windows	332263	299	1.00	1.00	1.00	1.00	1.00	1.00
ICMP PING speedera	46302	100	0.83	0.81	0.81	0.83	0.83	0.81
(http inspect) NON-RFC HTTP DELIMITER	13683	13653	0.41	0.54	0.57	0.30	0.37	0.50
(http inspect) OVERSIZE REQUEST-URI DIRECTORY	8811	8802	0.96	0.96	0.96	0.96	0.96	0.96
(http inspect) BARE BYTE UNICODE ENCODING	2426	2425	0.41	0.59	0.44	0.40	0.36	0.59
(http inspect) DOUBLE DECODING ATTACK	1447	1447	0.69	0.53	0.66	0.75	0.55	0.36
(http inspect) APACHE WHITESPACE (TAB)	1410	1409	0.47	0.60	0.53	0.59	0.40	0.59
average			0.70	0.72	0.71	0.70	0.66	0.67
(spp stream4) STEALTH ACTIVITY (unknown) detection	1800	1800	0.00	0.01	0.01	0.00	0.00	0.01
(snort decoder) Truncated Tcp Options	26495	25629	0.05	0.06	0.05	0.05	0.05	0.05

TABLE III
NUMBER OF FLOWS AND AVERAGE PRECISION PER RULE: BASELINE, DRIFT, AND SAMPLING

Alert message	Precision for recall of		Alert % for recall of	
	1.00	0.99	1.00	0.99
MS-SQL version overflow	1.00	1.00	3.0	2.9
CHAT AIM receive message	0.02	0.11	0.0	0.0
EXPLOIT ISAKMP first payload	0.02	0.03	0.9	0.6
ICMP PING				
CyberKit 2.2 Windows	1.00	1.00	0.1	0.0
ICMP PING speedera	0.02	0.83	0.5	0.0
(http inspect)				
NON-RFC HTTP DELIMITER	0.00	0.01	1.3	0.6
OVERSIZE REQUEST-URI DIR.	0.01	0.20	0.1	0.0
BARE BYTE UNICODE ENC.	0.00	0.00	1.1	0.4
DOUBLE DECODING ATTACK	0.00	0.00	1.8	0.4
APACHE WHITESPACE (TAB)	0.00	0.00	1.1	0.1

TABLE IV
PRECISION AND ALARM RATE AT HIGH RECALL FOR PAYLOAD RULES

D. Choosing an operating point

Choosing an appropriate operating threshold can be challenging. That is, above which confidence do we trigger an ML alarm? We have introduced precision, which is the proportion of ML alarms which are also Snort alarms. Another useful concept is the *recall*, which is the proportion of Snort alarms which are also ML alarms. A detector is perfect when both the precision and recall are 1, which in our case often happens for header and meta-information rules.

The story is more complicated for payload rules. The first 2 columns in Table IV report the precision for thresholds chosen to obtain a recall of 1 and 0.99, respectively. We see that we can get both high precision and recall only for the "MS-SQL version overflow attempt" and "ICMP PING CyberKit 2.2 Windows" rules. For all the rules whose average precision is below 0.7, the precision falls to near 0 for high recall values.

In cases where human post-processing is possible, high recall/low precision operating points can still be very useful, especially when the number of alarms is much lower than the total number of examples. As we can see in the last 2 columns in Table IV, even rules with comparatively low AP scores only raise alarms for a small percentage of flows to guarantee a recall of 1 or 0.9. For instance, the "APACHE WHITESPACE" rule, with an average precision below 0.6, can deliver a 0.99 recall while alerting on only 0.1% of the flows. While human examination of false positives is not possible in our case, we could imagine running Snort on the ML alarms, at a fraction of the cost of running Snort on all flows. This represents an interesting direction for future study but is beyond the scope of this paper.

Rule	base line	dest port	src port	num byte	num pack	dura tion	mean pack size	mean pack intval	TCP flag	IP serv type	dest IP local
Header											
ICMP Dest. Unreachable Comm. Admin. Prohib.	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
ICMP Destination Unreachable Comm. with Dest. Host Administratively Prohib.	0.99	0.00	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
ICMP Source Quench	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.61
average	1.00	0.00	1.00	1.00	1.00	0.99	0.99	1.00	1.00	1.00	0.87
Meta-information											
ICMP webtrends scanner	0.99	0.89	0.99	0.00	0.99	0.99	0.75	0.99	0.99	0.99	0.59
BAD-TRAFFIC data in TCP SYN packet	1.00	0.74	1.00	1.00	1.00	1.00	0.99	1.00	0.50	1.00	1.00
ICMP Large ICMP Packet	1.00	1.00	1.00	1.00	1.00	1.00	0.43	1.00	1.00	1.00	0.96
ICMP PING NMAP	1.00	1.00	1.00	1.00	1.00	1.00	0.02	1.00	1.00	1.00	0.50
SCAN FIN	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.24	0.99	0.99
average	1.00	0.92	1.00	0.80	1.00	1.00	0.64	1.00	0.75	1.00	0.81
Payload											
MS-SQL version overflow attempt	1.00	0.99	1.00	1.00	1.00	1.00	0.83	1.00	1.00	1.00	0.48
CHAT AIM receive message	0.61	0.64	0.61	0.49	0.55	0.42	0.33	0.54	0.29	0.51	0.51
EXPLOIT ISAKMP 1st payload length overflow	0.58	0.15	0.58	0.49	0.26	0.55	0.58	0.57	0.57	0.56	0.57
ICMP PING CyberKit 2.2 Windows	1.00	0.52	1.00	0.95	1.00	1.00	0.77	0.99	1.00	1.00	0.39
ICMP PING speedera	0.82	0.79	0.82	0.07	0.82	0.82	0.06	0.82	0.82	0.81	0.72
(http inspect) NON-RFC HTTP DELIM	0.48	0.02	0.34	0.15	0.47	0.24	0.22	0.32	0.22	0.42	0.42
average	0.75	0.52	0.72	0.52	0.68	0.67	0.46	0.71	0.65	0.72	0.52

TABLE V
THE IMPORTANCE OF EACH FEATURE TO A CLASSIFIER AS MEASURED BY THE AP IF THE FEATURE IS REMOVED DURING DETECTION

E. Detailed Analysis of ML Operation

In Section III we presented a taxonomy of Snort rules that distinguishes them according to the types of packet fields they access. "Payload rules" contain at least one predicate that inspects a packet's payload, "header rules" contain only predicates that can be exactly reproduced in a flow setting, and "meta rules" encompass all other Snort rules. Given enough training examples, a ML algorithm will be able to learn to perfectly classify flows according to header rules, whereas payload rules are generally much more challenging. As our results indicate, however, there are many meta rules that can be learned perfectly, and some payload rules as well.

We must delve deeper into the classifiers in order to understand the variability of detection accuracy within the payload and meta groups. Recall from Section V that a trained classifier is a *weighted* sum over each predicate. Since each predicate operates on a single feature (*e.g.*, TCP port, packet duration), this weight can provide intuition into the relative importance of this predicate to the classifier. For example, which of the destination port number or the flow duration is most important in order to correctly classify Slammer traffic? The standard way to measure the relative importance of each feature for a classifier is to measure the detection accuracy when the feature is removed. Thus, we train the classifier using all features, but then remove the given feature from consideration during classification: if detection accuracy goes down then clearly this feature was important. Table V reports the results of doing precisely this: each column reports the AP when the feature for that column is ignored during classification.

Table V demonstrates that Adaboost is able to correctly *interpret* (as opposed to merely mimic) many header rules by prioritizing the proper fields: the destination port, which encodes the ICMP code and type fields, is essential to each of the ICMP rules. Moreover, the meta rules that are learned well tend to inspect packet-header fields that are reported inexactly in flows, *e.g.*, packet payload size or TCP flags. The "SCAN FIN" rule is raised by Snort when only the FIN flag is set in a TCP packet. When we inspected the exact classifier generated by Adaboost (*i.e.*, this includes the chosen thresholds) for this rule, we found that Adaboost learns to raise this alarm whenever the aggregated TCP flags field in the flow header has a set FIN flag either by itself, combined with SYN, or combined with SYN and RST. As expected, no alarm is raised if the flow TCP flag field has FIN and ACK set.

Predicates that require access to packet payload information, on the other hand, cannot be reproduced in a flow setting whatsoever. For payload rules to be learned in a flow setting, therefore, the corresponding flow classifier must rely on some combination of (A) other predicates of the original Snort rule, and (B) entirely new predicates constructed by the ML algorithm to describe the packets/flows matching these rules. Table V contains several instances of each, and we will further investigate two (*viz.* "ICMP PING **CyberKit** 2.2 Windows" and "**MS-SQL** version overflow attempt") by inspecting the precise classifier generated by Adaboost.

The MS-SQL rule has several predicates, including one that matches a specific destination port number, one that inspects the size of the packet payload, and one that looks for a string pattern in the payload itself. Adaboost learns the first

predicate exactly, but learns a mean packet size predicate that is more precise than the Snort equivalent. That is, whereas Snort requires that the packet payload size must be greater than 100 bytes, Adaboost requires that the mean packet size should be 404 bytes, which in fact is the exact length of a SQL Slammer packet. (Indeed, the corresponding rule has been used in some cases to help identify Slammer traffic [1].) Combining this predicate and the destination port number, Adaboost learns this rule with high accuracy.

CyberKit is another payload rule that is learned by Adaboost with a high degree of accuracy. Table V shows that the important features for this classifier are (a) the destination port number, (b) the mean packet size, and (c) whether or not the target host is part of the configured local domain ("dest IP local"). The first and last of these features are a part of the Snort specification, but the mean packet size predicate is not. Adaboost tells us that flows that trigger this Snort alarm have a mean packet size between 92 and 100 bytes per packet.

The ability of ML algorithms to generate predicates independent of the original Snort specification is why ML algorithms are necessary over more rudimentary techniques. For example, a technique that identifies and translates only the flow and meta predicates from Snort rules (*i.e.*, those predicates that can be translated either exactly or approximately) would perform worse in the case of MS-SQL. While such simpler techniques would perform equally well for header rules, they would be ineffective for the majority of payload rules where only a ML approach has a chance to perform well.

IX. Implementation Scenarios and Architecture

We now describe how our approach could be exploited for flow alerting at network scale, in an architecture illustrated in Figure 3, whose components we now describe.

Flow Records: are collected from a cut set of interfaces across the network topology (edge and/or core) so that all traffic traverses at least one interface at which flow records are generated. The flow records are exported to a collector.
Packet Monitor / Alerter: a small number of packet monitors are located at sites chosen so as to see a representative mix of traffic. Each is equipped with a set of packet level rules which are applied to the observed packet stream. Alerts produced by the packet rules are forwarded to the ML trainer.
Machine Learning Trainer: correlates packet alerts with flows generated from the same traffic, and generates the set of flow level alerting rules. The rules are updated periodically, or in response to observed changes in traffic characteristics.
Runtime Flow Classifier: applies flow-level rules to all flow records, producing flow-level alerts.

Future Work. Section X investigates the scaling properties of computation required in this architecture. Now we consider the ML aspects that require further study. This paper used a single dataset for learning and testing. But the architecture requires that flow-level rules generated by ML on data gathered at a small number of sites can accurately alarm on flows measured at other sites. We propose to extend the study to multiple

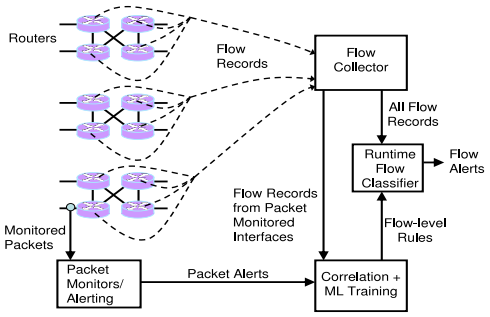

Fig. 3. Machine Learning Based Flow Alerting System

datasets gathered from different locations, training and testing on different datasets. One question is whether differences in the distribution of flow features such as duration, due, e.g., to different TCP dynamics across links of different speeds, could impair the accuracy of cross-site alarming. A further matter is to investigate the effect on detection accuracy if using packet sampled flow records for learning and classification.

X. Computational Efficiency

We analyze the computational speed of the three phases of our scheme: (i) correlation of flow records with Snort alarms prior to training; (ii) the ML phase; (iii) run-time classification of flows based on the learned flow rules. We combine analysis with experimental results to estimate the resources required for the architecture of Section IX. We consider two scenarios.

A: Scaling the interface rate: what resources are needed to perform correlation and ML at a higher data rate? We consider traffic equivalent to a full OC48 link (corresponding to a large ISP customer or data center). At 2.5Gbits/sec this is a scale factor 150 larger than our test dataset; we assume the numbers of positive and negative examples scale by the same factor.
B: Scaling classification across sites: Consider a set of network interfaces presenting traffic at rate of our data set; at 2MB/sec this represents medium sized ISP customers. The flow rules are learned from traffic on one of the interfaces. What resources are required to classify flows on the others?

A. Costs, Implementations, and Parallel Computation

We believe parallelization of correlation and learning steps is reasonable, since the cost is borne only once per learning site, compared with the cost deploying Snort to monitor at multiple locations at line rate. Parallelism for the classification step is far more costly, since its scale the resources required for at monitoring point. The implementations used here are not optimized, so the numerical values obtained are conservative.

B. Initial Correlation of Flow and Snort Data

Our prototype system can correlate flow records with Snort alarms at a rate if 275k flows per second on a 1.5 GHz Itanium 2 machine: about 15 minutes to correlate one week's data. Under our scaling scenario A, the hypothetical OC48 would require about 33 hours of computation on the same single

processor to correlate one week's data. This task is readily parallelized, the cost borne once prior to the learning stage.

C. Learning Step

The time taken for Adaboost or the Maxent algorithm we considered [17] to learn a given rule is proportional to the product of three quantities:

- the number of iterations N_i, which is fixed to the conservatively large number of 200 in our problem.
- the total number of +ve and -ve examples $N_e = n_- + n_+$
- the number of candidate weak classifiers N_c that Adaboost must consider

For numerical features, the number of weak classifiers is the number of boundaries that separate runs of feature values from positive and negative examples when laid out on the real line. This is bounded above by the twice the number n_+ of positive examples. We computed the dependence of N_c on data size for sampled subsets of the dataset; per rule, N_c scaled as n_+^α for some $\alpha < 1$. These behaviors suggest the following strategy to control computation costs for processing traffic while maintaining learning accuracy:

- Use all positive examples;
- Use at most fixed number n_-^0 of negative examples.

Limiting the number of negative examples does not impair accuracy since we still have more positive examples. Computation time is proportional to $N_i N_e N_c \leq 2N_i n_+(n_+ + n_-^0)$. While n_+ is much less than n_-^0—see Table I— computation time scales roughly linearly with the underlying data rate.

To see how this plays out in our hypothetical example, we take our dataset with 1 in 10 sampling of positive examples as representing the reference operating threshold. Hence, from Table I, there are roughly $n_-^0 = 4M$ unique negative examples. For n_+ we take the average number of unique positive examples per rule per week, namely 8861, the average of the second numerical column in Table III. Scaling to OC48 scales $n_+ \rightarrow 150n_+$ and hence $n_+(n_+ + n_-^0) \rightarrow 150n_+(150n_+ + n_-^0)$. Learning time increases by roughly a factor 200, lengthening the average time per rule from 10 minutes to 33 hours. Although this may seem large, it is conservative and likely unproblematic, since (i) it is far shorter that the data drift timescale of two weeks which should not depend on link speed, and can be reduced by (ii) optimized implementation; (iii) parallelization, once per learning site; and (iv) sampling the positive examples. Sampling may be desirable to control training time for rules with many positive examples, being precisely the rules for which sampling has the least impact on accuracy.

D. Classification Step

The number of predicates selected by Adaboost is typically around 100: the number of feature lookups and multiply-adds needed to test a rule. The same machine as above is able to apply these predicates, *i.e.* perform flow classification, at a rate of 57k flows/second. Our original dataset presented flows at a rate of about 530 flows/second, so this could nearly accommodate the 150 fold increase in flow rate in Scenario A, or classify flows from 100 interfaces in Scenario B.

XI. Conclusions

We proposed an ML approach to reproducing packet level alerts for anomaly detection at the flow level; Applying Snort rules to a single 4 week packet header trace, we found:

- Classification of flow-level rules according to whether they act on packet header, payload or meta-information is a good qualitative predictor of average precision.
- The ML approach is effective at discovering associations between flow and packet level features of anomalies and exploiting them for flow level alerting.
- Drift was largely absent at a timescale of two weeks, far longer than the few minutes required for learning.

We proposed an architecture to exploit this at network scale, and set out the steps for a proof of concept. We analyzed the computation complexity of our approach and argued that computation remains feasible at network scale. Although our study focused on single packet alarms produced by Snort, our approach could in principle we applied to learn from flow records alone, alarms generated by multipacket/flow events of the type monitored by Bro [19].

References

[1] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security and Privacy*, vol. 1, no. 4, pp. 33–39, 2003.

[2] "Snort," http://www.snort.org.

[3] "Cisco netflow. http://www.cisco.com/warp/public/732/netflow/."

[4] H. Madhyastha and B. Krishnamurthy, "A generic language for application-specific flow sampling," *Computer Communication Review*, April 2008.

[5] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *SIGCOMM '05*, 2005, pp. 217–228.

[6] T. Shon and J. Moon, "A hybrid machine learning approach to network anomaly detection," *Inf. Sci.*, vol. 177, no. 18, pp. 3799–3821, 2007.

[7] T. Ahmed, B. Oreshkin, and M. J. Coates, "Machine learning approaches to network anomaly detection," in *Proc. SysML*, 2007.

[8] A. Soule, K. Salamatian, and N. Taft, "Combining filtering and statistical methods for anomaly detection," in *IMC '05*, 2005, pp. 1–14.

[9] Y. Zhang, Z. Ge, A. Greenberg, and M. Roughan, "Network anomography," in *IMC '05*. New York, NY, USA: ACM, 2005, pp. 1–14.

[10] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *Internet Measurment Workshop*, 2002.

[11] L. Bernaille, R. Teixeira, and K. Salamatian, "Early application identification," in *Conference on Future Networking Technologies*, 2006.

[12] J. Erman, A. Mahanti, M. F. Arlitt, I. Cohen, and C. L. Williamson, "Offline/realtime traffic classification using semi-supervised learning," *Perform. Eval.*, vol. 64, no. 9–12, pp. 1194–1213, 2007.

[13] A. Moore and D. Zuev, "Internet traffic classification using bayesian analysis," in *Sigmetrics*, 2005.

[14] H. Jiang, A. W. Moore, Z. Ge, S. Jin, and J. Wang, "Lightweight application classification for network management," in *Proceedings of the SIGCOMM Workshop on Internet Network Management*, 2007.

[15] V. N. Vapnik, *Statistical Learning Theory*. John Wiley & Sons, 1998.

[16] R. E. Schapire and Y. Singer, "Improved boosting algorithms using confidence-rated predictions," *Machine Learning*, vol. 37, no. 3, pp. 297–336, 1999.

[17] M. Dudik, S. Phillips, and R. E. Schapire, "Performance Guarantees for Regularized Maximum Entropy Density Estimation," in *Proceedings of COLT'04*. Banff, Canada: Springer Verlag, 2004.

[18] N. Duffield, C. Lund, and M. Thorup, "Charging from sampled network usage," in *1st Internet Measurement Workshop*, 2001, pp. 245–256.

[19] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, pp. 2435–2463, Dec. 1999.