Mining Concept-Drifting Data Stream to Detect Peer to Peer Botnet Traffic

Technical Report UTDCS-05-08

Department of Computer Science

The University of Texas at Dallas

March 2008

Mohammad M. Masud, Jing Gao, Latifur Khan,

Jiawei Han and Bhavani Thuraisingham

Mining Concept-Drifting Data Stream to Detect Peer to Peer Botnet Traffic

Mohammad M. Masud * mehedy@utdallas.edu
Jing Gao † jinggao3@uiuc.edu
Latifur Khan * lkhan@utdallas.edu
Jiawei Han † hanj@cs.uiuc.edu
Bhavani Thuraisingham * bxt043000@utdallas.edu

ABSTRACT

We propose a novel stream data classification technique to detect Peer to Peer botnet. Botnet traffic can be considered as stream data having two important properties: infinite length and drifting concept. Thus, stream data classification technique is more appealing to botnet detection than simple classification technique. However, no other botnet detection approaches so far have applied stream data classification technique. We propose a multi-chunk, multi-level ensemble classifier based data mining technique to classify concept-drifting stream data. Previous ensemble techniques in classifying concept-drifting stream data use a single data chunk to train a classifier. In our approach, we train an ensemble of v classifiers from r consecutive data chunks. K of these v-classifier ensembles are used to build another level of ensemble. By introducing this multi-chunk, multi-level ensemble, we significantly reduce error compared to the single-chunk, single level ensemble. We have established the justification of using our algorithm theoretically. We have also tested our technique on both botnet traffic and simulated data, and obtained better detection accuracies compared to other published works.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms

Keywords

botnet, classification, data stream, ensemble

*Dept. of Computer Science, University of Texas at Dallas
†Dept. of Computer Science, University of Illinois at Urbana-Champaign

1. INTRODUCTION

Botnet is a network of compromised hosts or *bots*, under the control of a human attacker known as the *botmaster*. The botmaster can issue commands to the bots to perform malicious actions, such as recruiting new bots, launching coordinated DDoS attack against some hosts, stealing sensitive information from the bot machine, sending mass spam emails and so on. Thus, botnets have emerged as an enormous threat to the internet community.

The most prevailing botnets are the IRC-botnets [14], which have a centralized architecture. Besides, these botnets are usually very large and powerful, consisting of thousands of bots [13]. However, their enormous size and centralized architecture make them vulnerable to detection and demolition. Many approaches for detecting IRC botnets have been proposed recently [13, 7, 12, 10]. On the contrary, Peer to Peer (P2P) botnets are distributed, and much smaller than IRC botnets. So, they are more difficult to locate and take down. Most of the recent works in P2P botnet are in the analysis phase [9, 11, 8]. On the contrary, our work is aimed at *detecting* P2P botnets using network traffic mining.

Our botnet detection technique contributes to both data mining and network security communities, and distinguishes itself from the existing work from the following two perspectives. First, we apply stream data mining technique to detect botnet traffic. Although there have been some works in detecting botnet traffic using data mining [12], none of them applies stream data classification techniques. Stream data classification techniques could be effective to mining botnet traffic because of its two properties: infinite length and concept drift, to be explained shortly. Second, we propose an efficient technique for stream data classification, which is also an improvement over the previous stream data classification techniques.

Network traffic can be considered as a continuous flow of data stream produced by computing machines at an overwhelming rate. It is a major challenge to data mining community to be able to mine this ever-growing streaming data. There are two major problems related to stream data classification. First, it is impractical to store and use all the historical data for training, since it would require infinite storage and running time. Second, there may be concept-drift in the data. For example, in the context of botnets, the botmaster usually updates the bot software frequently, which may change the characteristics of botnet traffic, resulting in a concept drift in the data. However, the solutions to these two problems are related. If there is a concept-drift in the data, we need to refine our hypothesis to accommo-

date the new concept. Thus, most of the old data must be discarded from the training set. One of the main issues in mining concept-drifting streaming data is to choose the appropriate training instances to learn the evolving concept.

One approach is to select and store the training data that are most consistent with the current concept [3]. Some other approaches update the existing classification model when new data appears, such as the Very Fast Decision Tree (VFDT) [2] approach. Another approach is to use an ensemble of classifiers and update the ensemble every time a new data appears [17, 15]. As shown in [17, 15], the ensemble classifier is often more robust at handling unexpected changes and concept drifts. We follow this philosophy and propose a multi-chunk, multi-level ensemble classification algorithm, which could improve over the existing methods significantly.

A common approach in classifying stream data is to divide the stream data into equal sized chunks. We also follow this approach. However, instead of storing historical data, we store the trained classifiers. We always store an ensemble A of best K classifiers $\{A_1,...,A_K\}$. The ensemble A is actually a two-level ensemble. That is, each classifier A_i in the ensemble A is actually a collection (ensemble) of v classifiers. Thus, we build a hierarchy of ensembles, where A is at the top level of the hierarchy, and each of its children A_i is at the middle level. The lowest level (or the leaves) contains the actual classifiers. This is illustrated in figure 1.

As soon as a new data chunk appears, we train a new middle-level ensemble A_n. We update the top-level ensemble A by replacing a middle-level ensemble $A_i (1 \leq i \leq K)$ with the new ensemble A_n, if A_n has lower error rate than A_i. Our technique is different from previous ensemble techniques (e.g. [17]) in that we build classifiers from multiple data chunks, whereas previous techniques used to build one classifier per chunk. We use a window of r consecutive data chunks to build one middle-level ensemble. By introducing this multi-chunk multi-level ensemble, we reduce the expected error by a factor of rv over the single-chunk, single-level ensemble method. We prove the effectiveness of our approach both theoretically and empirically. Besides, in our approach, users have the flexibility to tune these two parameters (i.e., r, v) to improve performance up to a certain level.

We have several contributions. First, we propose a multi-chunk, multi-level ensemble technique that is a generalization over the existing single-chunk single-level ensemble techniques. Second, we prove the effectiveness of our technique theoretically. Finally, we apply our technique on synthetically generated data as well as on real botnet traffic, and achieve better detection accuracies than other stream data classification techniques. We believe that the proposed ensemble technique provides a powerful tool for network security and the results in the paper are very encouraging for the future use of stream data classification in botnet detection.

The rest of the paper is organized as follows: section 2 discusses related works, section 3 discusses our botnet traffic mining strategies, 4 discusses the classification algorithm and proves its effectiveness, section 5 discusses data collection, experimental setup, evaluation techniques, and results, and section 6 concludes with directions to future work.

2. RELATED WORK

Botnet research is being approached from three different perspectives: analysis, tracking, and detection. Barford et al. [1] present a comprehensive analysis about bots and botnets. Grizzard et al. [8] present an analysis of peer to peer botnets. An interesting botnet tracking and disruption technique is presented by Freiling et al [4]. Rajab et al. [13] describe a botnet tracking technique using IRC protocol.

A number of different IRC botnet detection techniques have been proposed recently. Goebel and Holz [7] detect bot infected machines by matching IRC nickname similarity. This is an effective detection technique, but requires payload for analysis. On the contrary, our method does not require payload. Karasaridis et al. [10] apply statistical properties of bot Command and Control (C&C) traffic to detect IRC botnet traffic and the C&C controller. Livadas et al. [12] apply data mining to detect C&C traffic passing through IRC channel. Although there are many IRC botnet detection techniques available, very few P2P botnet detection techniques have been seen so far. Besides, to our knowledge, there is no data mining based botnet traffic detection approach that apply stream data classification techniques. Our work is directed toward P2P botnet detection using a stream data classification technique.

There have been many works in stream data classification. There are two main approaches - single model classification, and ensemble classification. Single model classification techniques incrementally update their model with new data to accommodate concept drift [2]. These techniques usually require complex operations to modify the internal structure of the model. Besides, their prediction error may have large variance as only the most recent data is used to refine the model. On the contrary, ensemble techniques like Boosting [5] have the advantage that they can be more efficiently built than updating a single model. Besides, they observe higher accuracy than their single model counterpart [16]. Several researchers follow this appraoch for stream data mining [17, 15, 6].

We follow the same philosophy of Wang et al. [17] to keep an ensemble of the K best classifiers. Each time a new data chunk appears, Wang et al. build a classifier out of that chunk. If this classifier shows better accuracy than any of the K classifiers in the ensemble, then the new classifier replaces the old one. When classifying an instance, weighted voting among the classifiers in the ensemble is taken, where the weight of a classifier is inversely proportional to its error. There are three main differences between our approach and the approach of Wang et al. First, each classifier in our K classifier ensemble is actually another ensemble of v classifiers. Second, we train each classifier from r consecutive data chunks, rather than from a single chunk. Third, we use simple voting, rather than weighted voting. Thus, our multi-chunk, multi-level ensemble approach is a generalized form of the approach of Wang et al., and users have more freedom to optimize performance by choosing the appropriate values of these two parameters (i.e., r and v).

3. P2P BOTNET TRAFFIC MINING

P2P is the new emerging technology of botnets. As discussed earlier, these botnets are distributed, and smaller than the centralized, large IRC botnets. So, P2P botnets are not easy to detect and destroy. Examples of P2P bots are Nugache [11], Sinit [9], and Trojan.Peacomm [8]. These bots communicate with each other using the P2P protocol. We have identified several important characteristics of P2P

botnet traffic. In this section, we describe in details why these features are important, and how to extract them from the network traffic.

3.1 P2P botnet traffic characteristics

1. New connection establishment rate (NCR): Grizzrd et al. [8] found that Trojan.Peacomm establishes connection with its peers at a very high rate at the initial stage of infection. However, this rate diminishes after a while. Thus, a sudden increase in connection establishment rate is a good symptom of infection. So, we monitor the connection establishment rate to each server port.

2. Packet size (PS): Bots communicate with each other in their own language. Since all bots in a botnet are actually the same software, it is reasonable to assume that they would generate the same size packets to communicate with each other. In other words, the packet sizes generated by these bots should be within a certain range. But normal traffic should not have any specific range of packet sizes, since they are generated by different software. So, we monitor the packet sizes that are transmitted through each server port.

3. Upload/download bandwidth (UB/DB): The upload/download bandwidth in bot ports depend on the bot activity. Our observation of the bots reveals that they keep communicating with their peers at a certain rate at the time of spreading. Besides, it is reasonable to assume that at the time of combined attack, there will be a burst of traffic for a certain period of time. So, we monitor the upload/download bandwidth of each server port.

4. Arp request rate (ARP): When Nugache [11] bot infects a computer, it tries to contact with a initial set of 22 peers over the internet. The IP addresses of those peers are hard-coded in the bot's binary. According to our observation, the bot generates ARP requests for these IP addresses at a high rate (4-10 / minute) until it finds all of them. Thus, this is an important feature for bot infected machines.

5. ICMP echo reply rate(ICR): The Sinit bot [9] probes random IP addresses to find its peer. Thus, it generates a lot of "ICMP host unreachable" messages. So, the rate of ICMP host unreachable messages is also an important feature.

3.2 Monitoring botnet traffic

Our traffic monitoring system is host-based. That is, for each host, we monitor suspicious traffic going out or into the host. However, we do not need to monitor all the traffic. Rather, we can selectively choose which traffic should be monitored. Because, one important characteristic of P2P bots is that they must open server port(s) in the bot machine to communicate with its peers. Thus any malicious traffic due to the bot should appear on those server ports. Besides, P2P bots only use TCP or UDP ports. So, we need to monitor the only TCP or UDP server ports.

We detect server ports using a simple technique. A port is a server port if either of the followings are true:

1. It accepts a new connection, or
2. It is connected to multiple ports.

For example, a peer to peer bot called Nugache [11] opens a server port on TCP port 8. It accepts new connection to this port. Also, multiple clients may be connected to this port at the same time.

For each host, we create a list M of the server ports that are to be monitored. The ports in the list may be either local or remote server ports. Each port in M is continuously monitored. If it is a local port, then it is connected to multiple remote ports. Otherwise, it is connected to multiple host ports. In either case, we monitor the traffic going into and out of the host ports. In addition to monitoring the server ports, we also monitor the ARP and ICR features that are related to the host itself.

All the five features of botnet traffic that have been mentioned are extracted from packet header. We choose to monitor packet header, rather than payload, because payload monitoring has several drawbacks. First, there is always a privacy issue. Second, payload data may not always be available for training or testing. Finally, with respect to P2P botnet, payload is sometimes obfuscated, or encrypted. Monitoring an obfuscated/encrypted channel may not reveal any useful information.

3.3 Feature extraction

Extracting NCR: For each monitored port, we create a histogram of new connection establishment rate having B_1 bins $\{NCR[0], ..., NCR[B_1-1]\}$, representing B_1 ranges of values, namely, $\{[0,1),[1,2),..., [B_1-2, B_1-1), [B_1-1, \infty)\}$. We update these bins every s seconds as follows: if the total number new of connections made after the last update is i, then we increment the bin counter $NCR[min(B_1-1, i)]$.

Extracting PS: For each monitored port, we create a histogram of packet sizes having B_2 bins $\{PS[0], ..., PS[B_2-1]\}$ representing B_2 ranges of values, namely, $\{ [0, 2^5), [2^5, 2^6) ,..., [2^{B_2+2}, 2^{B_2+3}) ,[2^{B_2+3}, \infty) \}$ bytes. When a new packet of size ps arrives, we increment the bin counter corresponding to the range where ps belongs.

Extracting UB/DB: For each monitored port, we create a histogram of upload bandwidth having B_3 bins $\{UB[0], ..., UB[B_3-1]\}$ and a histogram of download bandwidth having B_3 bins $\{DB[0], ..., DB[B_3-1]\}$ representing B_3 ranges of byte transfer rates in every s seconds, namely, $\{ [0, 2^5), ... , [2^{B_3+3}, \infty) \}$ bytes. If the total number of bytes uploaded(downloaded) since last update is $ub(db)$, then we increment the bin counter corresponding to the range where $ub(db)$ belogs.

Extracting ARP: We create a histogram of Arp request rates having B_4 bins $\{ARP[0], ..., ARP[B_4-1]\}$ representing B_4 ranges of Arp request rates in s seconds, namely, $\{[0,1) ,..., [B_4-1, \infty)\}$. We update the bin counters in every s seconds.

Extracting ICMP: To extract this feature, we create a histogram of ICMP host unreachable message rate having B_5 bins $\{ICR[0], ..., ICR[B_5-1]\}$ representing B_5 ranges of Arp request rates in every s seconds, namely, $\{[0,1) ,..., [B_5-1, \infty)\}$. We update the bins every s seconds.

4. A MULTI-CHUNK MULTI-LEVEL ENSEMBLE (MCE) FOR CONCEPT DRIFTING STREAM DATA

The data stream is divided into equal-sized chunks. We always store only the most recent best K classifiers in our ensemble. Let the ensemble of classifiers be $A= \{A_1, A_2, ... , A_K\}$. Each classifier in the ensemble A_i is actually an ensemble of v classifiers trained with r consecutive data chunks. That is, $A_i= \{A_{i(1)}, A_{i(2)}, ... , A_{i(v)}\}$, where $A_{i(j)}$'s

are trained with r consecutive data chunks. This ensemble hierarchy and the training process is illustrated in figures 1 and 2. We will refer to our approach as the "Multi-Chunk Multi-Level Ensemble (MCE)" approach.

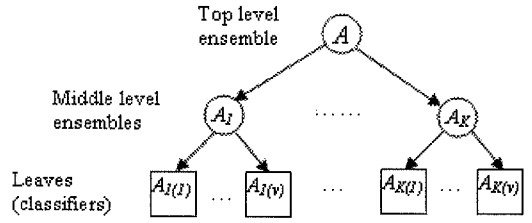

Figure 1: Hierarchy of ensembles in MCE

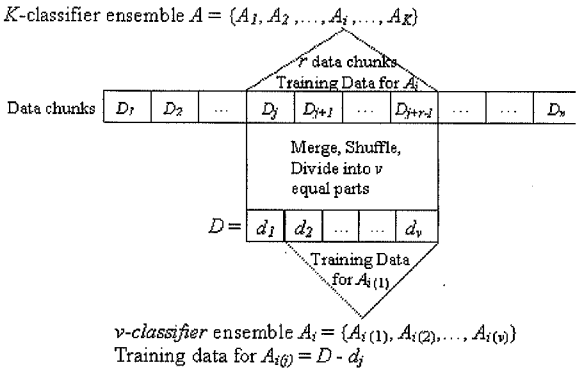

Figure 2: Illustration: how data chunks are used to build an ensemble with MCE

4.1 Ensemble updating algorithm (MCE)

The algorithm is illustrated in Algorithm 1.

Description of the algorithm: Let D_n be the most recent data chunk that has been labeled. In line 3 of the algorithm, we compute the error of each ensemble $A_i \in A$ on D_n. Note that A is the top level ensemble, and each A_i is a middle-level ensemble. Let D=$\{D_n, D_{n-1}, \ldots, D_{n-r+1}\}$, i.e, the most recent r data chunks including D_n. In line 6, we randomly divide D into v equal parts = $\{d_1, \ldots, d_v\}$, such that roughly, all the parts have the same number of positive and negative examples. In lines 7-11, we build a new middle-level ensemble A_n having v classifiers = $\{A_{n(1)}, A_{n(2)}, \ldots, A_{n(v)}\}$, where each classifier $A_{n(j)}$ is trained with the dataset D - $\{d_j\}$. We compute the expected error of the ensemble A_n by testing each classifier $A_{n(j)}$ on d_j and averaging their error. Finally, on line 12, we select the best K classifiers from the $K+1$ classifiers $A_n \cup A$.

4.2 Error reduction using multi-level ensemble and multi-chunk training

As explained in the algorithm, we build an ensemble of K concepts A, where each concept A_i is itself an ensemble of v concepts. Thus, we build a hierarchy of ensembles where A is at the root of the hierarchy having K children A_i; and each A_i has v children $A_{i(j)}$. A test data x is classified in a bottom-up approach. At first, x is tested with the lowest

Algorithm 1 Updating the classifier ensemble

```
1: /* Let D_n be the newest data chunk Let A be the
   current ensemble of best K classifiers */
2: for each ensemble A_i ∈ A do
3:    Test A_i on D_n and compute its expected error
4: end for
5: Let D = {D_n ∪ D_{n-1} ... ∪ D_{n-r+1}}
6: Divide D into v equal disjoint parts {d_1, d_2, ..., d_v}
7: for j=1 to v do
8:    A_{n(j)} ← Train a classifier with training data D-d_j
9:    Test A_{n(j)} on its test data d_j
10:   Update the expected error of A_n
11: end for
12: A ← best K classifiers from A_n ∪ A based on expected
    error
```

level classifiers $A_{i(j)}$, and the output is passed to the parent. Then each A_i outputs the majority class of its children and the output is passed to the root. Finally, A outputs the majority class of its children. This is illustrated in algorithm 2. In the next few paragraphs, we prove that MCE reduces the

Algorithm 2 Classifying an instance with MCE

```
1: /* Let x be a test instance and A be the current en-
   semble of best K classifiers */
2: for each ensemble A_i ∈ A do
3:    for each classifier A_{i(j)} ∈ A_i do
4:       c_{i(j)} ← class prediction of x according to A_{i(j)}
5:    end for
6:    c_i ← majority class among all c_{i(j)}
7: end for
8: prediction ← majority class among all c_i
```

expected error in classifying concept-drifting data streams by a factor of rv over other approaches that use only one data chunk for training a single classifier (i.e., $r = 1$, $v = 1$), which will be referred to henceforth as "Single-Chunk Single-Level Ensemble (SCE)" approach.

Given an instance x, the posterior probability distribution of class c is $p(c|x)$. For a two-class classification problem, c=+ or −. According to Tumer and Ghosh [16], a classifier is trained to learn a function f that approximates this posterior probability:

$$f(x) = p(c|x) + \eta(x) \tag{1}$$

where $\eta(x)$ is the error of f relative to $p(c|x)$, which is the error in addition to Bayes error (added error). According to [16], the expected added error can be obtained from the following equation:

$$Error = \frac{\sigma_\eta^2}{s} \tag{2}$$

where σ_η^2 is the variance of η, and s is the difference between the derivatives of $p(+|x)$ and $p(-|x)$, which is independent of the learned classifier.

If we average the outputs of the classifiers in a K-classifier ensemble, then according to [16] , the ensemble output would be:

$$f_A = \frac{1}{K}\sum_{i=i}^{K} f_{A_i}(x) = p(c|x) + \eta_A(x) \tag{3}$$

where f_A is the output of the ensemble of the K classifiers A (i.e., MCE), $f_{A_i}(x)$ is the output of the ith classifier A_i, and $\eta_A(x)$ is the average error of all classifiers, given by:

$$\eta_A(x) = \frac{1}{K}\sum_{i=1}^{K}\eta_{A_i}(x) \tag{4}$$

where $\eta_{A_i}(x)$ is the added error of the ith classifier in the ensemble. Assuming the error variances are independent, the variance of $\eta_A(x)$ is given by:

$$\sigma^2_{\eta_A(x)} = \frac{1}{K^2}\sum_{i=1}^{K}\sigma^2_{\eta_{A_i}(x)} \tag{5}$$

where $\sigma^2_{\eta_{A_i}(x)}$ is the variance of $\eta_{A_i}(x)$. In order to simplify the notation, we would denote $\sigma^2_{\eta_{A_i}(x)}$ with $\sigma^2_{A_i}$.

Let C be the the ensemble of K classifiers $\{C_1, C_2, \ldots, C_K\}$,where each C_i is a classifier trained using a single data chunk, i.e., the SCE approach. The following two theorems prove that MCE reduces error over SCE by a factor of rv.

THEOREM 1. *Let σ^2_C be the error variance of SCE. If there is no concept drift, then the error variance of MCE is at most $1/rv$ times of that of SCE. i.e.,*

$$\sigma^2_A \leq \frac{1}{rv}\sigma^2_C$$

PROOF. Suppose B_i be a single classifier, as opposed to the ensemble of v classifiers A_i, trained on the same r consecutive data chunks. According to [16], if the error variances of the individual classifiers $A_{i(j)}$, $1 \leq j \leq v$ are independent, then the ensemble error is $1/v$ times the error of the single classifier B_i. In other words,

$$\sigma^2_{A_i} = \frac{1}{v}\sigma^2_{B_i} \tag{6}$$

where $\sigma^2_{B_i}$ is the error variance of B_i. If there is no concept drift, then according to Wang et al. [17], a classifier trained on r consecutive data chunks may reduce the error of the single classifiers trained on a single data chunk by a factor of r. So, it follows that

$$\sigma^2_{B_i} = \frac{1}{r^2}\sum_{j=i}^{r+i-1}\sigma^2_{C_j} \tag{7}$$

where $\sigma^2_{B_i}$ is the error variance of classifier B_i, trained using data chunks $\{D_i \cup D_{i+1} \ldots \cup D_{i+r-1}\}$ and $\sigma^2_{C_j}$ is the error variance of C_j, trained using a single data chunk D_j. Combining (5),(6),(7) and simplifying, we get:

$$\begin{aligned}
\sigma^2_A &= \frac{1}{K^2 v}\sum_{i=1}^{K}\frac{1}{r^2}\sum_{j=i}^{r+i-1}\sigma^2_{C_j} \\
&= \frac{1}{K^2 r^2 v}\sum_{i=1}^{K}\sum_{j=i}^{r+i-1}\sigma^2_{C_j} \\
&\leq \frac{1}{rv}\left(\frac{1}{K^2}\sum_{i=1}^{K}\sigma^2_{C_i}\right), K >= r \\
&= \frac{1}{rv}\sigma^2_C
\end{aligned}$$

□

However, if there is concept drift, then the assumption (7) may not be valid. Suppose P_d is the maximum error introduced due to concept drift. That is, every time a new data chunk appears, the classification error of each classifier C_i is incremented by a factor of $(1 + P_d)$. Let $C_j, j \in \{i, i+1, ..., i+r-1\}$ be the classifier trained with a single data chunk D_j in an window of r consecutive chunks. Then according to our refined assumption,

$$\hat{\sigma}^2_{C_j} = (1 + P_d)^{(i+r-1)-j}\sigma^2_{C_j} \tag{8}$$

where $\hat{\sigma}^2_{C_j}$ is the actual error of the jth classifier C_j (with concept drift), when the last data chunk in the window, D_{i+r-1} appears. Our second theorem deals with error reduction in the presence of concept drift.

THEOREM 2. *Let $\hat{\sigma}^2_A$ be the error variance of MCE in the presence of concept drift, σ^2_C be the error variance of SCE, and P_d be the drifting probability defined above. Then $\hat{\sigma}^2_A$ is bounded by:*

$$\hat{\sigma}^2_A \leq \frac{(1 + P_d)^{r-1}}{rv}\sigma^2_C$$

PROOF. Replacing $\sigma^2_{C_j}$ with $\hat{\sigma}^2_{C_j}$ we get

$$\begin{aligned}
\hat{\sigma}^2_A &= \frac{1}{K^2}\sum_{i=1}^{K}\frac{1}{r^2}\sum_{j=i}^{r+i-1}\hat{\sigma}^2_{C_j} \\
&= \frac{1}{K^2 r^2 v}\sum_{i=1}^{K}\sum_{j=i}^{r+i-1}(1 + P_d)^{(i+r-1)-j}\sigma^2_{C_j} \\
&\leq \frac{1}{K^2 r^2 v}\sum_{i=1}^{K}(1 + P_d)^{r-1}\sum_{j=i}^{r+i-1}\sigma^2_{C_j} \\
&= \frac{(1 + P_d)^{r-1}}{K^2 r^2 v}\sum_{i=1}^{K}\sum_{j=i}^{r+i-1}\sigma^2_{C_j} \\
&\leq \frac{(1 + P_d)^{r-1}}{K^2 rv}\sum_{i=1}^{K}\sigma^2_{C_i}, r > 0 \\
&= \frac{(1 + P_d)^{r-1}}{rv}\sigma^2_C
\end{aligned}$$

□

Therefore, we would achieve a reduction of error provided that

$$\frac{(1 + P_d)^{r-1}}{rv} \leq 1 \qquad or, E_R \leq 1 \tag{9}$$

Where E_R is the ratio of MCE error to SCE error in the presence of concept-drift. As we increase r and v, the relative error keeps decreasing upto a certain point. After that, it becomes flat or starts increasing. Next, we analyze the effect of parameters r and v on error reduction, in the presence of concept-drift.

4.3 Upper bounds of r and v

For a given value of v, r can only be increased up to a certain value. After that, increasing r actually hurts the performance of our algorithm, because inequality (9) is violated. Figure 3 (left chart) shows the relative error E_R for $v = 1$, and different values of P_d, for increasing r. It is clear from the graph that for lower values of P_d, increasing r reduces the relative error by a greater margin. However,

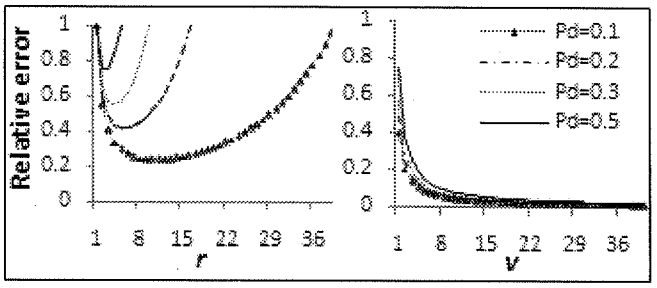

Figure 3: Error reduction by increasing r and v

in any case, after a certain value of r, E_R becomes greater than one. This is because, when we increase r, the numerator $(1+P_d)^{r-1}$ increases much faster than the denominator. Although it may not possible to know the actual value of P_d from the data, we may determine the optimal value of r experimentally. In our experiments, we found that for smaller chunk-sizes, higher values of r works better, and vice versa. However, the best performance-cost trade-off is found for r=2 or 3. We have used r=2 in most of our experiments.

We may also be able to reduce error if we increase v. Figure 3 (right chart) shows the the relative error E_R for $r = 2$, and different values of P_d, for increasing v. We see that the relative error keeps decreasing as we increase v. This is true for any value of P_d. However, after certain value of v, the rate of improvement gradually diminishes. Thus, we may choose an optimal value of v that best suites the performance-cost trade-off. Because, increasing v also linearly increases the running time. From our experiments, we obtained the best trade-off for v=5. This value also seems to be the optimal one from Figure 3. We used this value in most of our experiments.

4.4 Time complexity of MCE

Let m be the size of the data stream and n be the total number of data chunks. Then our time complexity is $O(Kvm + nvf(rm/n))$, where $f(z)$ is the time to build a classifier on a training data of size z. Since v is constant, the complexity becomes $O(Km + nf(rm/n))$ $=O(n.(Ks + f(rs))$, where s is the size of one data chunk. It should be mentioned here that the time complexity of the approach by Wang et al. [17] is $O(n.(Ks + f(s))$. Thus, the actual running time of MCE would be at most a constant factor (rv times) higher than that of Wang et al. But at the same time, we also achieve an error reduction by a factor of rv. Actually, users are free to choose the most suitable values of r and v to achieve a better performance within the specified time constraint.

4.5 Multi-level voting (MCE) versus only leaf level voting (MCE2)

In MCE, an instance is classified using two-level voting. First, it is classified with all the leaf-level classifiers $A_{i(j)}$. Then each middle-level ensemble A_i outputs the majority class of its children's prediction. Finally, the top-level ensemble A takes a majority voting of its children's outputs and predicts the final class. However, we may reduce the voting level by directly taking a majority of all the Kv leaf-level classifier outputs. When we use this voting, we mention our algorithm as MCE2. We found that MCE2 performs slightly better than MCE in most cases, although theoretically, MCE2 and MCE should have the same performance. We report the results of MCE2 in the results section.

5. EXPERIMENTS

We evaluate our proposed method on both synthetic data and botnet traffic, and in comparison with several baseline methods, our approach shows better performance in terms of classification accuracy.

5.1 Data sets and experimental setup

Synthetic data generation.

Synthetic data are generated with drifting concepts [17]. Concept-drifting data can be generated with a moving hyperplane. The equation of a hyperplane is as follows:

$$\sum_{i=1}^{d} a_i x_i = a_0 \tag{10}$$

If $\sum_{i=1}^{d} a_i x_i \leq a_0$, then an example is negative, otherwise it is positive. Each example is a randomly generated d-dimensional vector $\{x_1, ..., x_d\}$, where $x_i \in [0, 1]$. Weights $\{a_1, ..., a_d\}$ are also randomly initialized with a real number in the range [0, 1]. The value of a_0 is adjusted so that roughly the same number of positive and negative examples are generated. This can be done by choosing $a_0 = \frac{1}{2}\sum_{i=1}^{d} a_i$ We also introduce noise randomly by switching the labels of p% of the examples, where p=5 is set in our experiments.

There are several parameters that simulate concept drift. Parameter m specifies the percent of total dimensions whose weights are involved in changing, and it is set to 20%. Parameter t specifies the magnitude of the change in every N examples. In our expeiments, t is varied from 0.1 to 1.0, and N is set to 1000. $s_i, i \in \{1, ..., d\}$ specifies the direction of change for each weight. Weights change continuously, i.e., a_i is adjusted by $s_i.t/N$ after each example is generated. There is a possibility of 10% that the change would reverse direction after every N examples are generated. We generate a total of 250,000 records and generate four different datasets having chunk sizes 250, 500, 750, and 1000, respectively. So, the 250 chunk-size dataset contains 1000 chunks, the 500 chunk-size dataset contains 500 chunks and so on.

Botnet data collection.

We set up a controlled environment for collecting real botnet traffic. The system consists of four virtual machines running on top of a Windows XP host operating system. The host machine is an Intel Pentium-IV 3.2GHz dual core processor with 2GB RAM and 150GB Hard Disk. Each virtual machine is running Windows XP with 256 MB virtual RAM and 8GB virtual Hard Disk space. Each virtual machine runs a P2P bot named Nugache [11]. These machines constitute an isolated subnetwork such that each of them is visible from the others. This is similar to a small network of computers where each computer has the same subnet mask, and all are connected to a single hub/switch.

We run Windump (http://www.winpcap.org/windump/) from all machines to collect packet traces. It is the windows version of the "tcpdump" utility of UNIX. After running windump from each machine, we run the bot binary and collect packet traces for forty hours. We also collect

forty-hour packet traces from normal machines that are connected to the internet. We collect samples every 60 seconds (i.e., s=60), and update the bins of each histogram as explained in section 3.3. The bin sizes are chosen as follows: B_1=11, B_2=11, B_3=20, B_4=11, B_5=11. These are experimental values and does not guarantee any optimal performance. Ten consecutive samples are combined to make a data point.

So, for every monitored port, we obtain a data point in every ten minutes consisting of the bin counter values of all six histograms: {NCR[0] ,..., NCR[10], PS[0] ,..., PS[10], UB[0] ,..., UB[19], DB[0], ..., DB[19] , ARP[0] , ... ,ARP[10], ICR[0], ... ,ICR[10] }. Also, after every ten minutes, the bin counters are initialized to zero to build a new data point. We label the data point as "benign" or "malicious" based on whether a bot is using the corresponding port or not. We generate four different datasets having chunk sizes of 30 minutes, 60 minutes, 90 minutes, and 120 minutes, respectively. So, the 30 minutes chunk data contains total 80 equal sized chunks, the 60 minutes chunk data contains 40 chunks, and so on.

Baseline methods.

For classification, we use the "Weka" package, freely available from "http://www.cs.waikato.ac.nz/ml/weka/". We apply three different classifiers - J48 decision tree, Ripper, and Bayes Net. In order to compare with other techniques, we implement the followings:

MCE2: This is our modified MCE algorithm.

BestK: This is an SCE approach, where the ensemble of the best K classifiers is used. This ensemble is created by storing all the classifiers seen so far, and selecting the best K of them based on expected error. A test instance is tested using simple voting.

Last: In this case, we only keep the last trained classifier, trained on a single data chunk. It can be considered a SCE approach with $K = 1$. Once a new data is obtained, we train a new classifier and discard the old one.

Wang: The method implemented by Wang et al. [17]. This is also an SCE approach.

All: This is also an SCE approach. In this case, we create an ensemble of all the classifiers seen so far, and the new data chunk is with this ensemble by simple voting among the classifiers.

5.2 Performance study

In this section, we compare the results of all the five techniques, $MCE, Wang, BestK, All$ and $Last$. As soon as a new data chunk appears, we test each of these ensembles /classifiers on the new data, and update its accuracy, false positive, and false negative rate. In all the results shown here, we fix the parameter values of v=5, and r=2.

Figure 4 shows the error rate for K=2,4,6 and 8 of each method averaged over four different chunk sizes (250, 500, 750, and 1000). Here decision tree is used as the base learner. It is evident that $MCE2$ has the lowest error of all. Besides, we see that the error of $MCE2$ is lower for higher values of K. This is desired because higher values of K means larger ensemble, and more error reduction. However, accuracy does not improve much after $K = 8$. $Wang$ and $BestK$ also show similar characteristic. All and $Last$ do not depend on K, so their error remains the same for any K. Figure 5 shows the error for four different chunk-sizes of each of

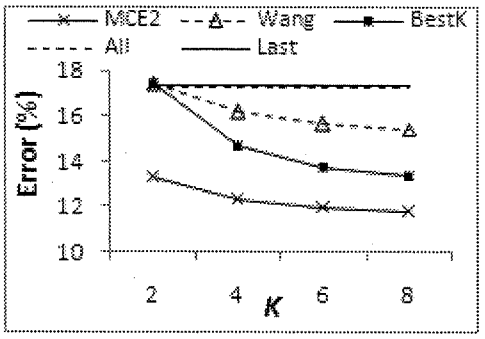

Figure 4: Error vs K on synthetic data

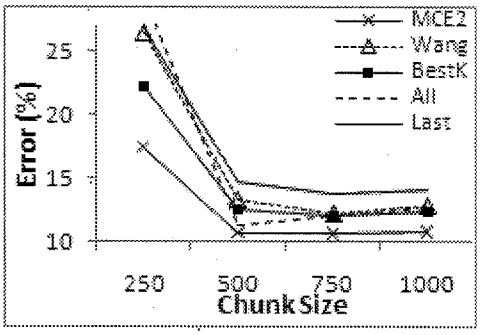

Figure 5: Error vs chunk size on synthetic data

these methods (also using decision tree) averaged over different values of K (2,4,6,8). Again, $MCE2$ has the lowest error of all. Besides, we see that the error of $MCE2$ is lower for larger chunk-sizes. This is desired because larger chunk-size means more training data for a classifier. However, after chunk-size=500, error does not reduce significantly. This indicates that this chunk-size is probably optimal for training. Other methods also have the similar characteristics.

Table 1 reports the error of decision tree learner on synthetic data, for different values of K and chunk-sizes. Tables 2 and 3 report the same of Bayes Net and Ripper algorithm, respectively. The columns denoted by M_2, W_2 and B_2 represents $MCE2$, $Wang$ and $BestK$, respectively, for K=2. Other columns have similar interpretations. In all three tables, we see that MCE2 has the lowest error for all values of K.

Figure 6 shows the effects of increasing r and v on the synthetic data for MCE2. The left chart of figure 6(a) shows the errors for different values of r for a fixed value of v (=5) and K (=8). We see that the highest reduction in error occurs when r is increased from 1 to 3. Note that $r = 1$ means single chunk training. Thus, multi-chunk training helps to reduce error. However, no significant reduction is observed for higher values of r. This follows from our analysis of parameter r on concept-drifting data, that increasing r will not help reducing error after a certain point. However, the running time keeps increasing, as shown in the left chart of figure 6(b). The best trade-off between running time and error occurs for r=2. The right charts of figure 6(a) and 6(b) show a similar trend for parameter v. Note that $v = 1$ is the base case, i.e., the single level ensemble approach, and $v > 1$ is the two-level ensemble approach. Thus, the multi-

Table 1: Error of different approaches on synthetic data using decision tree

Chunk size	M_2	W_2	B_2	M_4	W_4	B_4	M_6	W_6	B_6	M_8	W_8	B_8	*All*	*Last*
250	19.3	26.8	26.9	17.3	26.5	22.1	16.6	26.3	20.4	16.2	26.1	19.5	29.2	26.8
500	11.4	14.8	14.7	10.6	13.2	12.4	10.3	12.7	11.6	10.2	12.4	11.3	11.3	14.7
750	11.1	13.9	13.9	10.6	12.1	11.9	10.3	11.5	11.4	10.3	11.3	11.2	15.8	13.8
1000	11.4	14.3	14.3	10.7	12.8	12.2	10.5	12.2	11.7	10.3	11.9	11.4	12.6	14.1

Table 2: Error of different approaches on synthetic data using Bayes Net

Chunk size	M_2	W_2	B_2	M_4	W_4	B_4	M_6	W_6	B_6	M_8	W_8	B_8	*All*	*Last*
250	20.3	29.3	25.4	18.7	29.0	22.8	18.2	28.9	21.9	17.9	28.8	21.7	32.1	27.1
500	12.7	14.2	14.2	12.4	13.3	13.3	12.3	13.2	13.1	12.1	13.1	12.9	12.9	14.6
750	13.1	14.4	14.4	12.9	13.6	13.5	12.9	13.3	13.3	12.9	13.2	13.3	16.7	15.1
1000	13.0	14.2	14.2	12.7	13.3	13.5	12.6	13.2	13.4	12.5	13.4	13.1	13.6	14.4

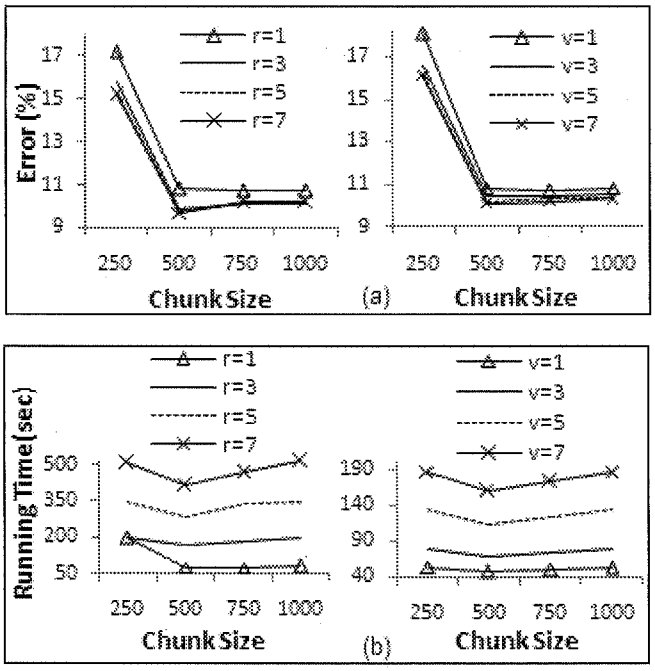

Figure 6: Effects of increasing parameters r and v: (a) on error, (b) on running time

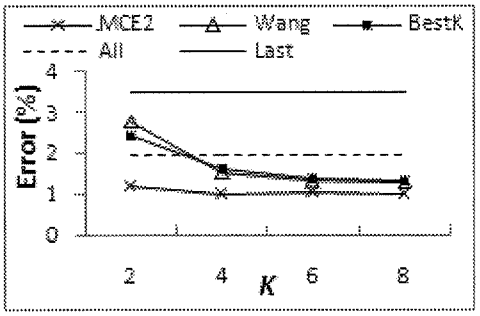

Figure 7: Error vs K on botnet data

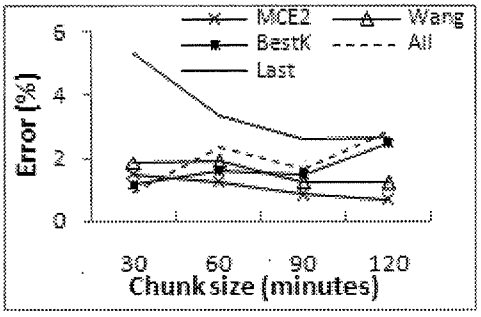

Figure 8: Error vs chunk size on botnet data

level ensemble has lower error than the single-level approach. However, there is no real improvement after $v = 5$, although the running time keeps increasing at the same rate. This result also follows from our analysis of the upper bounds of v, explained in section 4.3. Here we may choose v=3 or 5 as the best trade-off between time and error.

Figure 7 shows the error for K=2,4,6 and 8, on botnet data using decision tree as a base learner, averaged over four different chunk sizes. Again $MCE2$ has the lowest error of all, and *Last* has the highest error. Besides, the trend is similar to the synthetic data, because the error of $MCE2$ is lower for higher values of K. Figure 8 shows the error with decision tree for chunk-sizes 30-120 minutes, averaged over four different values of K. For the 30-minute chunk-size, *BestK* and *All* has slightly better performance (less than 0.2%) than $MCE2$. However, for all other chunk sizes, $MCE2$ has the lowest error. $MCE2$, *Last* and *Wang* show the same tendency that error is reduced with increasing chunk size.

Figure 9 (left chart) shows the running time of each method using decision tree on synthetic data for $K = 8, v = 5$ and $r = 2$ and varying chunk sizes. We see that $MCE2$ has the longest running time, although it is within 5 times of the running time of *Wang*. This actually follows from our complexity analysis that the running time of MCE would be at most rv times the running time of *Wang*. We also find the running time of $MCE2$ increases with chunk size. This is because training time goes up for larger chunk sizes. Running times of *Wang* and *Last* are almost co-incidental. However, it is interesting to note that running times of *BestK* and *All* decrease with increasing chunk size. This is so because as chunk size increases, the total number of classifiers produced by the stream decreases. So, *BestK* has to do less work to update its ensembles, and *All* has to do less work to classify

Table 3: Error of different approaches on synthetic data using Ripper

Chunk size	M_2	W_2	B_2	M_4	W_4	B_4	M_6	W_6	B_6	M_8	W_8	B_8	*All*	*Last*
250	19.2	26.5	26.0	17.6	26.2	22.4	17.1	26.0	21.3	16.8	25.9	20.9	30.4	26.3
500	11.5	14.2	13.9	10.8	13.0	12.3	10.6	12.6	11.8	10.5	12.5	11.5	11.6	14.1
750	11.0	13.4	13.3	10.6	12.1	12.0	10.5	11.7	11.6	10.5	11.5	11.5	15.7	13.3
1000	11.1	13.8	13.7	10.6	12.5	12.3	10.3	12.1	11.9	10.2	11.9	11.8	12.6	13.6

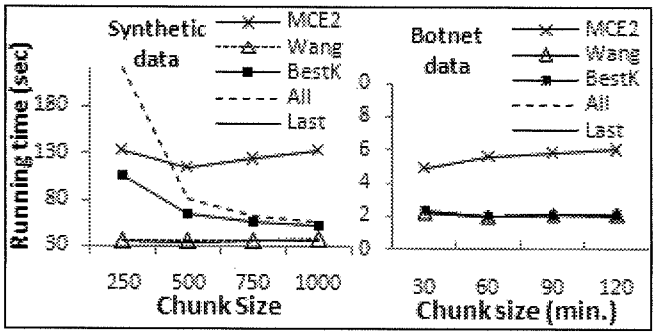

Figure 9: Chunk size vs running time

an instance. Figure 9 (right chart) shows the running time of all five approaches on botnet data for $K = 8, v = 5$ and $r = 2$ and varying chunk sizes. This graph also has similar behavior as the synthetic data. We would like to mention again here that there is always a performance-cost trade-off for $MCE2$. Users have the flexibility to choose either better performance or lower running time just by changing the parameters r and v.

6. CONCLUSION

We have two important contributions. First, we have applied stream data classification technique to detect P2P botnet traffic. Botnet traffic has two important properties of stream data: infinite length, and drifting concept, which make stream data classification techniques very appealing to botnet detection. However, no other botnet detection approaches have applied stream data classification techniques yet. Second, we have introduced a multi-chunk multi-level ensemble method (MCE) for classifying concept-drifting stream data. Our MCE approach is more effective in correctly classifying concept-drifting stream data over other existing ensemble based stream data classification techniques.

Our ensemble approach keeps the best K concepts, where each concept itself is an ensemble of v classifiers, trained with r consecutive data chunks. It is a generalization over previous ensemble approaches that used to train a single classifier from a single data chunk. By introducing this multi-chunk multi-level ensemble, we have reduced error by a factor of $r.v$, which has been proved theoretically. Besides, we have tested our approach on both synthetic data and real botnet data, and obtained better performance compared to other approaches.

In future we would like to apply pruning to reduce the number of classifiers in the ensemble, keeping the same prediction accuracy. Besides, we would also like to apply our technique on the classification and model evolution of other real streaming data.

7. REFERENCES

[1] P. Barford and V. Yegneswaran. An inside look at botnets. In *Special Workshop on Malware Detection, Advances in Information Security, Springer*, 2006.

[2] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. SIGKDD*, pages 71–80, 2000.

[3] W. Fan. Systematic data selection to mine concept-drifting data streams. In *Proc. KDD*, pages 128–137, 2004.

[4] F. Freiling, T. Holz, and W. G. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proc. of ESORICS, Milan, Italy.*, September 2005.

[5] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Proc. ICML*, pages 148–156, 1996.

[6] J. Gao, W. Fan, and J. Han. On appropriate assumptions to mine data streams. In *Proc. ICDM*, 2007.

[7] J. Goebel and T. Holz. Rishi: Identify bot contaminated hosts by irc nickname evaluation. In *Usenix/Hotbots '07 Workshop*, 2007.

[8] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *Usenix/Hotbots '07 Workshop*, 2007.

[9] L. T. I. Group. Sinit p2p trojan analysis. lurhq. *http://www.lurhq.com/sinit.html*, 2004.

[10] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *Usenix/Hotbots '07 Workshop*, 2007.

[11] R. Lemos. Bot software looks to improve peerage. *http://www.securityfocus.com/news/11390*, 2006.

[12] C. Livadas, B. Walsh, D. Lapsley, and T. Strayer. Using machine learning techniques to identify botnet traffic. In *2nd IEEE LCN Workshop on Network Security (WoNS'2006)*, November 2006.

[13] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proc. of the 6th ACM SIGCOMM on Internet Measurement Conference (IMC)*, 2006.

[14] B. Saha and A. Gairola. Botnet: An overview. *CERT-In White Paper CIWP-2005-05*, 2005.

[15] M. Scholz and R. Klinkenberg. An ensemble classifier for drifting concepts. In *Proc. ICML/PKDD Workshop in Knowledge Discovery in Data Streams.*, 2005.

[16] K. Tumer and J. Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8(304):385–403, 1996.

[17] H. Wang, W. Fan, P. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proc. KDD*, 2003.