Cloud-Based Malware Detection for Evolving Data Streams

MOHAMMAD M. MASUD, TAHSEEN M. AL-KHATEEB,
and KEVIN W. HAMLEN, University of Texas at Dallas
JING GAO, University of Illinois at Urbana-Champaign
LATIFUR KHAN, University of Texas at Dallas
JIAWEI HAN, University of Illinois at Urbana-Champaign
BHAVANI THURAISINGHAM, University of Texas at Dallas

Data stream classification for intrusion detection poses at least three major challenges. First, these data streams are typically infinite-length, making traditional multipass learning algorithms inapplicable. Second, they exhibit significant concept-drift as attackers react and adapt to defenses. Third, for data streams that do not have any fixed feature set, such as text streams, an additional feature extraction and selection task must be performed. If the number of candidate features is too large, then traditional feature extraction techniques fail.

In order to address the first two challenges, this article proposes a multipartition, multichunk ensemble classifier in which a collection of v classifiers is trained from r consecutive data chunks using v-fold partitioning of the data, yielding an ensemble of such classifiers. This multipartition, multichunk ensemble technique significantly reduces classification error compared to existing single-partition, single-chunk ensemble approaches, wherein a single data chunk is used to train each classifier. To address the third challenge, a feature extraction and selection technique is proposed for data streams that do not have any fixed feature set. The technique's scalability is demonstrated through an implementation for the Hadoop MapReduce cloud computing architecture. Both theoretical and empirical evidence demonstrate its effectiveness over other state-of-the-art stream classification techniques on synthetic data, real botnet traffic, and malicious executables.

Categories and Subject Descriptors: H.2.8 [**Database Applications**]: Data mining; D.4.6 [**Security and Protection**]: Invasive software

General Terms: Algorithms, Security

Additional Key Words and Phrases: Data mining, malware detection, data streams, malicious executable, n-gram analysis

ACM Reference Format:
Masud, M. M., Al-Khateeb, T. M., Hamlen, K. W., Gao, J., Khan, L., Han, J., and Thuraisingham, B. 2011. Cloud-based malware detection for evolving data streams. ACM Trans. Manag. Inform. Syst. 2, 3, Article 16 (October 2011), 27 pages.
DOI = 10.1145/2019618.2019622 http://doi.acm.org/10.1145/2019618.2019622

This material is based on work supported by the AFOSR under awards FA9450-08-1-0260 and FA9950-10-1-0088 and by NASA under award 2008-00867-01.
Authors' addresses: M. M. Masud (corresponding author), T. M. Al-Khateeb, and K. W. Hamlen, Computer Science Department, University of Texas at Dallas, Dallas, TX; email: mehedy@utdallas.edu; J. Gao, Computer Science Department, University of Illinois at Urbana-Champaign; L. Khan, Computer Science Department, University of Texas at Dallas, Dallas, TX; J. Han, Computer Science Department, University of Illinois at Urbana-Champaign; B. Thuraisingham, Computer Science Department, University of Texas at Dallas, Dallas, TX.

DOI 10.1145/2019618.2019622 http://doi.acm.org/10.1145/2019618.2019622

1. INTRODUCTION

Malware is a potent vehicle for many successful cyber attacks every year, including data and identity theft, system and data corruption, and denial of service; it therefore constitutes a significant security threat to many individuals and organizations. The average direct malware cost damages worldwide per year from 1999 to 2006 have been estimated at $14 billion USD [Computer Economics, Inc. 2007]. This includes labor costs for analyzing, repairing, and disinfecting systems, productivity losses, revenue losses due to system loss or degraded performance, and other costs directly incurred as the result of the attack. However, the direct cost does not include the prevention cost, such as antivirus software, hardware, and IT security staff salary, etc. Aside from these monetary losses, individuals and organizations also suffer identity theft, data theft, and other intangible losses due to successful attacks.

Malware includes viruses, worms, Trojan horses, time and logic bombs, botnets, and spyware. A number of techniques have been devised by researchers to counter these attacks; however, the more successful the researchers become in detecting and preventing the attacks, the more sophisticated malicious code appears in the wild. Thus, the arms race between malware authors and malware defenders continues to escalate. One popular technique applied by the antivirus community to detect malicious code is *signature detection*. This technique matches untrusted executables against a unique telltale string or byte pattern known as a *signature*, which is used as an identifier for a particular malicious code. Although signature detection techniques are widely used, they are not effective against zero-day attacks (new malicious code), polymorphic attacks (different encryptions of the same binary), or metamorphic attacks (different code for the same functionality) [Crandall et al. 2005]. There has therefore been a growing need for fast, automated, and efficient detection techniques that are robust to these attacks. This article describes a data mining technique that is dedicated to automated generation of signatures to defend against these kinds of attacks.

1.1. Malware Detection as a Data Stream Classification Problem

The problem of detecting malware using data mining [Schultz et al. 2001; Kolter and Maloof 2004; Masud et al. 2008a] involves classifying each executable as either *benign* or *malicious*. Most past work has approached the problem as a static data classification problem, where the classification model is trained with fixed training data. However, the escalating rate of malware evolution and innovation is not well suited to static training. Detection of continuously evolving malware is better treated as a *data stream* classification problem. In this paradigm, the data stream is a sequence of executables in which each data point is one executable. The stream is *infinite-length*. It also observes *concept-drift* as attackers relentlessly develop new techniques to avoid detection, changing the characteristics of the malicious code. Similarly, the characteristics of benign executables change with the evolution of compilers and operating systems.

Data stream classification is a major area of active research in the data mining community, and requires surmounting at least three challenges: First, the storage and maintenance of potentially unbounded historical data in an infinite-length, concept-drifting stream for training purposes is infeasible. Second, the classification model must be adapted continuously to cope with concept-drift. Third, if there is no predefined feature space for the data points in the stream, new features with high discriminating power must be selected and extracted as the stream evolves, which we call *feature evolution*.

Solutions to the first two problems are related. Concept-drift necessitates refinement of the hypothesis to accommodate the new concept; most of the old data must be discarded from the training set. Therefore, one of the main issues in mining concept-drifting data streams is the selection of training instances adequate to learn the evolving concept. Solving the third problem requires a feature selection process that is ongoing, since new and more powerful features are likely to emerge and old features are likely to become less dominant as the concept evolves. If the feature space is large, then the running time and memory requirements for feature extraction and selection becomes a bottleneck for the data stream classification system.

One approach to addressing concept-drift is to select and store the training data that are most consistent with the current concept [Fan 2004]. Other approaches, such as Very Fast Decision Trees (VFDTs) [Domingos and Hulten 2000], update the existing classification model when new data appear. However, past work has shown that ensemble techniques are often more robust for handling unexpected changes and concept-drifts [Wang et al. 2003; Scholz and Klinkenberg 2005; Kolter and Maloof 2005]. These maintain an ensemble of classifiers and update the ensemble when new data appear.

We propose a multipartition, multichunk ensemble classification algorithm that generalizes existing ensemble methods. The generalization leads to significantly improved classification accuracy relative to existing single-partition, single-chunk ensemble approaches when tested on real-world data streams. The ensemble in our approach consists of Kv classifiers, where K is a constant and v is the number of partitions, to be explained shortly.

Our approach divides the data stream into equal sized *chunks*. The chunk size is chosen so that all data in each chunk fits into the main memory. Each chunk, when *labeled*, is used to train classifiers. Whenever a new data chunk is labeled, the ensemble is updated as follows. We take the r most recent labeled consecutive data chunks, divide these r chunks into v *partitions*, and train a classifier with each partition. Therefore, v classifiers are trained using the r consecutive chunks. We then update the ensemble by choosing the best Kv classifiers (based on accuracy) among the newly trained v classifiers and the existing Kv classifiers. Thus, the total number of classifiers in the ensemble remains constant. Our approach is therefore parameterized by the number of partitions v, the number of chunks r, and the ensemble size K.

Our approach does not assume that new data points appearing in the stream are immediately labeled. Instead, it defers the ensemble updating process until labels for the data points in the latest data chunk become available. In the meantime, new unlabeled data continue to be classified using the current ensemble. Thus, the approach is well suited to applications in which misclassifications solicit corrected labels from an expert user or other source. For example, consider the online credit card fraud detection problem. When a new credit card transaction takes place, its class (*fraud* or *authentic*) is predicted using the current ensemble. Suppose a fraudulent transaction is misclassified as *authentic*. When the customer receives the bank statement, he identifies this error and reports it to the authority. In this way, the actual labels of the data points are obtained and the ensemble is updated accordingly.

1.2. Cloud Computing for Malware Detection

If the feature space of the data points is not fixed, a subproblem of the classification problem is the extraction and selection of features that describe each data point. As in prior work (e.g., Kolter and Maloof [2004]), we use binary n-grams as features for malware detection. However, since the total number of possible n-grams is prohibitively large, we judiciously select n-grams that have the greatest discriminatory power. This selection process is ongoing; as the stream progresses, newer n-grams appear that dominate the older n-grams. These newer n-grams replace the old in our model in order to identify the best features for a particular period.

Naïve implementation of the feature extraction and selection process can be both time- and storage-intensive for large datasets. For example, our previous work [Masud et al. 2008a] extracted roughly a quarter billion *n*-grams from a corpus of only 3500 executables. This feature extraction process required extensive virtual memory (with associated performance overhead), since not all of these features could be stored in main memory. Extraction and selection required about 2 hours of computation and many gigabytes of disk space for a machine with a quad-core processor and 12GB of memory. This is despite the use of a purely static dataset; when the dataset is a dynamic stream, extraction and selection must recur, resulting in a major bottleneck. In this article we consider a much larger dataset of 105 thousand executables for which our previous approach is insufficient.

We therefore propose a scalable feature selection and extraction solution that leverages a cloud computing framework [Dean and Ghemawat 2008]. We show that depending on the availability of cluster nodes, the running time for feature extraction and selection can be reduced by a factor of m, where m is the number of nodes in the cloud cluster. The nodes are machines with inexpensive commodity hardware. Therefore, the solution is also cost effective as high-end computing machines are not required.

1.3. Contributions and Organization of the Article

Our contributions can therefore be summarized as follows. We propose a generalized multipartition, multichunk ensemble technique that significantly reduces the expected classification error over existing single-partition, single-chunk ensemble methods. A theoretical analysis justifies the effectiveness of the approach. We then formulate the malware detection problem as a data stream classification problem and identify drawbacks of traditional malicious code detection techniques relative to our data mining approach. We propose a scalable and cost-effective solution to this problem using a cloud computing framework. Finally, we apply our technique to synthetically generated data as well as real botnet traffic and real malicious executables, achieving better detection accuracy than other stream data classification techniques. The results show that our proposed ensemble technique constitutes a powerful tool for intrusion detection based on data stream classification.

The rest of the article is organized as follows: Section 2 discusses related works. Section 3 discusses the classification algorithm and proves its effectiveness analytically. Section 4 then describes the feature extraction and selection technique using cloud computing for malware detection, and Section 5 discusses data collection, experimental setup, evaluation techniques, and results. Section 6 discusses several issues related to our approach, and finally, Section 7 summarizes our conclusions.

2. RELATED WORK

Our work is related to both malware detection and stream mining. Both are discussed in this section.

Traditional *signature-based* malware detectors identify malware by scanning untrusted binaries for distinguishing byte sequences or *features*. Features unique to malware are maintained in a *signature database*, which must be continually updated as new malware is discovered and analyzed. Traditionally, signature databases have been manually derived, updated, and disseminated by human experts as new malware appears and is analyzed. However, the escalating rate of new malware appearances and the advent of self-mutating, polymorphic malware over the past decade have made manual signature updating less practical. This has led to the development of automated data mining techniques for malware detection (e.g., Kolter and Maloof [2004], Schultz et al. [2001], Masud et al. [2008a], and Hamlen et al. [2009]) that are capable of automatically inferring signatures for previously unseen malware.

Data-mining-based approaches analyze the content of an executable and classify it as malware if a certain combination of features are found (or not found) in the executable. These malware detectors are first trained so that they can generalize the distinction between malicious and benign executables, and thus detect future instances of malware. The training process involves feature extraction and model building using these features. Data-mining-based malware detectors differ mainly on how the features are extracted and which machine learning technique is used to build the model. The performance of these techniques largely depends on the quality of the features that are extracted.

Schultz et al. [2001] extract DLL call information (using *GNU binutils*) and character strings (using *GNU strings*) from the headers of Windows PE executables, as well as 2-byte sequences from the executable content. The DLL calls, strings, and bytes are used as features to train models. Models are trained using two different machine learning techniques, RIPPER [Cohen 1996] and Naïve Bayes (NB) [Michie et al. 1994], to compare their relative performances. Kolter and Maloof [2004] extract binary *n*-gram features from executables and apply them to different classification methods, such as *k*-Nearest Neighbor (KNN) [Aha et al. 1991], NB, Support Vector Machines (SVM) [Boser et al. 1992], decision trees [Quinlan 2003], and boosting [Freund and Schapire 1996]. Boosting is applied in combination with various other learning algorithms to obtain improved models (e.g., boosted decision trees). Our previous work on data-mining-based malware detection [Masud et al. 2008a] extracts binary *n*-grams from the executable, assembly instruction sequences from the disassembled executables, and DLL call information from the program headers. The classification models used in this work are SVM, decision tree, NB, boosted decision tree, and boosted NB.

Hamsa [Li et al. 2006] and Polygraph [Newsome et al. 2005] apply a simple form of data mining to generate worm signatures automatically using binary *n*-grams as features. Both identify a collection of *n*-grams as a worm signature if they appear only in malicious binaries (i.e., positive samples) and never in benign binaries. This differs from the traditional data mining approaches already discussed (including ours) in two significant respects: First, Polygraph and Hamsa limit their attention to *n*-grams that appear only in the malicious pool, whereas traditional data mining techniques also consider *n*-grams that appear in the benign pool to improve the classification accuracy. Second, Polygraph and Hamsa define signature matches as simply the presence of a set of *n*-grams, whereas traditional data mining approaches build classification models that match samples based on both the presence and absence of features. Traditional data mining approaches therefore generalize the approaches of Polygraph and Hamsa, with corresponding increases in power.

Almost all past work has approached the malware detection problem as a static data classification problem in which the classification model is trained with fixed training data. However, the rapid emergence of new types of malware and new obfuscation strategies adopted by malware authors introduces a dynamic component to the problem that violates the static paradigm. We therefore argue that effective malware detection must be increasingly treated as a data stream classification problem in order to keep pace with attacks.

Many existing data stream classification techniques target infinite-length streams that exhibit concept-drift (e.g., Aggarwal et al. [2006], Wang et al. [2003], Yang et al. [2005], Kolter and Maloof [2005], Hulten et al. [2001], Fan [2004], Gao et al. [2007], Hashemi et al. [2009], and Zhang et al. [2009]). All of these techniques adopt a one-pass incremental update approach, but with differing approaches to the incremental updating mechanism. Most can be grouped into two main classes: single-model incremental approaches and hybrid batch-incremental approaches.

Single-model incremental updating involves dynamically updating a single model with each new training instance. For example, decision tree models can be incrementally updated with incoming data [Hulten et al. 2001]. In contrast, hybrid batch-incremental approaches build each model from a batch of training data using a traditional batch learning technique. Older models are then periodically replaced by newer models as the concept drifts [Wang et al. 2003; Bifet et al. 2009; Yang et al. 2005; Fan 2004; Gao et al. 2007]. Some of these hybrid approaches use a single model to classify the unlabeled data (e.g., Yang et al. [2005] and Chen et al. [2008]) while others use an ensemble of models (e.g., Wang et al. [2003] and Scholz and Klinkenberg [2005]). Hybrid approaches have the advantage that model updates are typically far simpler than in single-model approaches; for example, classifiers in the ensemble can simply be removed or replaced. However, other techniques that combine the two approaches by incrementally updating the classifiers within the ensemble can be more complex [Kolter and Maloof 2005].

Accuracy Weighted classifier Ensembles (AWE) [Wang et al. 2003; Scholz and Klinkenberg 2005] are an important category of hybrid-incremental updating ensemble classifiers that use weighted majority voting for classification. These divide the stream into equal-sized chunks, and each chunk is used to train a classification model. An ensemble of K such models classifies the unlabeled data. Each time a new data chunk is labeled, a new classifier is trained from that chunk. This classifier replaces one of the existing classifiers in the ensemble. The replacement victim is chosen by evaluating the accuracy of each classifier on the latest training chunk. These ensemble approaches have the advantage that they can be built more efficiently than a continually updated single model, and they observe higher accuracy than their single-model counterparts [Tumer and Ghosh 1996].

Our ensemble approach is most closely related to AWE, but with a number of significant differences. First, we apply multipartitioning of the training data to build v classifiers from that training data. Second, the training data consists of r consecutive data chunks (i.e., a multichunk approach) rather than from a single chunk. We prove both analytically (in Section 3.2) and empirically (in Section 5.4) that both of these enhancements, that is, multipartitioning and multichunk, significantly reduces ensemble classification error. Third, when we update the ensemble, v classifiers in the ensemble are replaced by v newly trained classifiers. The v classifiers that are replaced may come from different chunks; thus, although some classifiers from a chunk may have been removed, other classifiers from that chunk may still remain in the ensemble. This differs from AWE, in which removal of a classifier means total removal of the knowledge obtained from one whole chunk. Our replacement strategy also contributes to error reduction, as discussed in Section 5.4. Finally, we use simple majority voting rather than weighted voting, which is more suitable for data streams, as shown in Gao et al. [2007]. Thus, our multipartition, multichunk ensemble approach is a more generalized and efficient form of that implemented by AWE.

Our proposed work extends our previously published work [Masud et al. 2009]. Most existing data stream classification techniques, including our previous work, assumes that the feature space of the data points in the stream is fixed. However, in some cases, such as text data, this assumption is not valid. For example, when features are words, the feature space cannot be fully determined at the start of the stream since new words appear frequently. In addition, it is likely that much of this large lexicon of words has low discriminatory power, and is therefore best omitted from the feature space. It is therefore more effective and efficient to select a subset of the candidate features for each data point. This feature selection must occur incrementally as newer, more discriminating candidate features arise and older features become outdated. Therefore, feature extraction and selection should be an integral part of data stream classification.

In this article, we propose an efficient and scalable feature extraction and selection technique using a cloud computing framework [Zhao et al. 2009; Dean and Ghemawat 2008]. This approach supersedes our previous work in that it considers the real

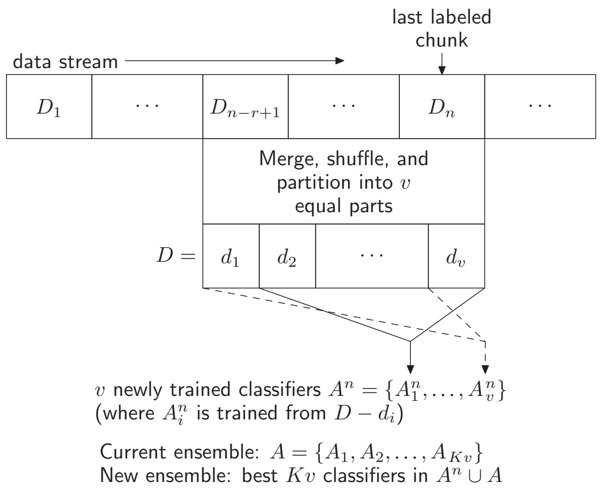

last labeled chunk
data stream
D_1
$\cdots$
D_{n-r+1}
$\cdots$
D_n
$\cdots$
Merge, shuffle, and partition into v equal parts
$D =$
d_1
d_2
$\cdots$
d_v
v newly trained classifiers $A^n = \{A_1^n, \ldots, A_v^n\}$
(where A_i^n is trained from $D - d_i$)
Current ensemble: $A = \{A_1, A_2, \ldots, A_{Kv}\}$
New ensemble: best Kv classifiers in $A^n \cup A$

Fig. 1. Building an ensemble from data chunks.

challenges in data stream classification that occur when the feature space cannot be predetermined. This facilitates application of our technique to the detection of real malicious executables from a large, evolving dataset, showing that it can detect newer varieties of malware as malware instances evolve over time.

3. TECHNICAL APPROACH

Our *Extended, MultiPartition, multi-Chunk (EMPC)* ensemble learning approach maintains an ensemble $A = \{A_1, A_2, \ldots, A_{Kv}\}$ of the most recent, best Kv classifiers. Each time a new data chunk D_n arrives, it tests the data chunk with the ensemble A. The ensemble is updated once chunk D_n is labeled. The classification process uses simple majority voting.

3.1. Ensemble Construction and Updating

The ensemble construction and updating process is illustrated in Figure 1 and summarized in Algorithm 1.

Lines 1–3 of the algorithm compute the error of each classifier $A_i \in A$ on chunk D_n, where D_n is the most recent data chunk that has been labeled. Let D be the data of the most recently labeled r data chunks, including D_n. Line 5 randomly partitions D into v equal parts $\{d_1, \ldots, d_v\}$ such that all the parts have roughly the same class distributions.

Lines 6–9 train a new batch of v classifiers, where each classifier A_j^n is trained with dataset $D - d_j$. The error of each classifier $A_j^n \in A^n$ is computed by testing it on its corresponding test data d_j. Finally, line 10 selects the best Kv classifiers from the $Kv + v$ classifiers in $A^n \cup A$ based on the errors of each classifier computed in lines 2 and 8. Note that any subset of the nth batch of v classifiers may be selected for inclusion in the new ensemble.

3.2. Error Reduction Analysis

As explained in Algorithm 1, we build an ensemble A of Kv classifiers. A test instance x is classified using a majority vote of the classifiers in the ensemble. We use simple majority voting rather than weighted majority voting (refer to Wang et al. [2003]), since simple majority voting has been theoretically proven the optimal choice for data

ALGORITHM 1: Updating the classifier ensemble

```
Input: $\{D_{n-r+1}, \ldots, D_n\}$: the $r$ most recently labeled data chunks
    $A$: the current ensemble of best $Kv$ classifiers
Output: an updated ensemble $A$
 1: for each classifier $A_i \in A$ do
 2:     $e(A_i) \leftarrow$ error of $A_i$ on $D_n$ // test and compute error
 3: end for
 4: $D \leftarrow \cup_{j=n-r+1}^{n} D_j$
 5: Partition $D$ into equal parts $\{d_1, d_2, \ldots, d_v\}$
 6: for $j = 1$ to $v$ do
 7:     $A_j^n \leftarrow$ newly trained classifier from data $D - d_j$
 8:     $e(A_j^n) \leftarrow$ error of $A_j^n$ on $d_j$ // test and compute error
 9: end for
10: $A \leftarrow$ best $Kv$ classifiers from $A^n \cup A$ based on computed error $e(.)$
```

streams [Gao et al. 2007]. Weighted voting can be problematic in these contexts because it assumes that the distribution of training and test data are the same. However, in data streams, this assumption is violated because of concept-drift. Simple majority voting is therefore a better alternative. Our experiments confirm this in practice, obtaining better results with simple rather than weighted majority voting.

The following argument shows that EMPC can further reduce the expected error in classifying concept-drifting data streams compared to *Single-Partition, single-Chunk (SPC)* approaches, which use only one data chunk for training a single classifier (i.e., $r = v = 1$). Intuitively, there are two main reasons for the error reduction. First, the training data per classifier is increased by introducing the multichunk concept. Larger training data naturally leads to better trained model, reducing the error. Second, rather than training only one model from the training data, we partition the data into v partitions, and train one model from each partition. This further reduces error because the mean expected error of an ensemble of v classifiers is theoretically v times lower than that of a single classifier [Tumer and Ghosh 1996]. Therefore, both the multichunk and multipartition strategy contribute to error reduction.

Given an unlabeled test instance x, the posterior probability distribution of class a is $p(a|x)$. A classifier is trained to learn a function $f^a(\cdot)$ that approximates this posterior probability

$$f^a(x) = p(a|x) + \beta^a + \eta^a(x), \tag{1}$$

where β^a is the bias of the classifier, and $\eta^a(x)$ is the variance of the classifier given input x [Tumer and Ghosh 1996]. These biases and variances are the *added error* beyond the Bayes error. We limit our attention to the variance term because we would like to analyze the error introduced by training the classifiers on different data chunks. (The bias may differ with varying classifiers if they are trained with different learning algorithms, but in our case the same learning algorithm is used in different data chunks.)

The expected error of a classifier is given by

$$Err = \frac{\sigma^2_{\eta^a(x)}}{s}, \tag{2}$$

where $\sigma^2_{\eta^a(x)}$ is the variance of $\eta^a(x)$, and s is independent of the learned classifier [Tumer and Ghosh 1996]. Let $C = \{C_1, \ldots, C_K\}$ be an ensemble of K classifiers, where each classifier C_i is trained from a single data chunk (i.e., C is an SPC ensemble). If we average the outputs of the classifiers in a K-classifier ensemble, then the ensemble output f_C^a becomes

$$f_C^a = \frac{1}{K}\sum_{i=i}^{K} f_{C_i}^a(x) = p(a|x) + \eta_C^a(x), \tag{3}$$

where f_C^a is the output of the ensemble C, $f_{C_i}^a(x)$ is the output of the ith classifier C_i, and $\eta_C^a(x)$ is the average error of all classifiers, given by

$$\eta_C^a(x) = \frac{1}{K}\sum_{i=1}^{K} \eta_{C_i}^a(x). \tag{4}$$

Here, $\eta_{C_i}^a(x)$ is the added error of the ith classifier in the ensemble. Assuming the error variances are independent, the variance of $\eta_C^a(x)$ is given by

$$\sigma^2_{\eta_C^a(x)} = \frac{1}{K^2}\sum_{i=1}^{K} \sigma^2_{\eta_{C_i}^a(x)} = \frac{1}{K}\bar{\sigma}^2_{\eta_C^a(x)}. \tag{5}$$

where $\sigma^2_{\eta_{C_i}^a(x)}$ is the variance of $\eta_{C_i}^a(x)$, and $\bar{\sigma}^2_{\eta_C^a(x)}$ is the common variance. In order to simplify the notation, we denote $\sigma^2_{\eta_{C_i}^a(x)}$ as $\sigma^2_{C_i(x)}$.

Let $A = \{A_1, A_2, \ldots, A_{Kv}\}$ be the ensemble of Kv classifiers, where each classifier A_i is trained using r consecutive data chunks (i.e., the EMPC approach). The following lemma proves that EMPC reduces error over SPC by a factor of rv when the outputs of the classifiers in the ensemble are independent.

LEMMA 1. *Let $\sigma^2_{C(x)}$ be the error variance of SPC. If there is no concept-drift, and the errors of the classifiers in the ensemble A are independent, then the error variance of EMPC is a fraction $(rv)^{-1}$ of that of SPC.*

$$\sigma^2_{A(x)} = \frac{1}{rv}\sigma^2_{C(x)} \tag{6}$$

PROOF. Each classifier $A_i \in A$ is trained on r consecutive data chunks. If there is no concept-drift, then a classifier trained on r consecutive chunks may reduce the error of the single classifier trained on a single chunk by a factor of r [Wang et al. 2003]. It follows that

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

where $\sigma^2_{A_i(x)}$ is the error variance of classifier A_i trained using data $\bigcup_{j=i}^{r+i-1} D_i$, and $\sigma^2_{C_j(x)}$ is the error variance of C_j, trained using a single data chunk D_j. Combining Eqs. (5) and (7) and simplifying, we obtain

$$\sigma^2_{A(x)} = \frac{1}{K^2v^2}\sum_{i=1}^{Kv}\left(\frac{1}{r^2}\sum_{j=i}^{r+i-1}\sigma^2_{C_j(x)}\right),$$
$$= \frac{1}{K^2v^2r}\sum_{i=1}^{Kv}\left(\frac{1}{r}\sum_{j=i}^{r+i-1}\sigma^2_{C_j(x)}\right),$$

$$= \frac{1}{K^2v^2r}\sum_{i=1}^{Kv}\bar{\sigma}^2_{C_i(x)} = \frac{1}{Krv}\left(\frac{1}{Kv}\sum_{i=1}^{Kv}\bar{\sigma}^2_{C_i(x)}\right),$$

$$= \frac{1}{Krv}\bar{\sigma}^2_{C(x)} = \frac{1}{rv}\sigma^2_{C(x)}, \tag{8}$$

where $\bar{\sigma}^2_{C_i(x)}$ is the common variance of $\sigma^2_{C_j(x)}$ for $j \in [i, i+r-1]$, and $\bar{\sigma}^2_{C(x)}$ is the common variance of $\bar{\sigma}^2_{C_i(x)}$ for $i \in [1, Kv]$. □

Although it may appear from Lemma 1 that by making r arbitrarily large, error can be made arbitrarily small, this is not true in practice because of concept-drift. We will elaborate this issue in Lemma 3.

Lemma 1 does not consider the effects of possible correlation among the classifiers in the ensemble. Such correlations are potentially significant in our approach because each set of v classifiers is trained from r consecutive data chunks. As a result, each pair of these v classifiers has overlapping training data. This can result in considerable correlations between the classifiers, which must be taken into consideration in any realistic analysis of the expected error reduction of the system. The following lemma accounts for these effects by introducing a mean correlation term δ to the analysis of Lemma 1.

LEMMA 2. *Let $\sigma^2_{C(x)}$ be the error variance of SPC. If there is no concept-drift, then the error variance of EMPC is a fraction $(v-1)/(rv)$ of that of SPC, where $v > 1$.*

$$\sigma^2_{A(x)} \leq \frac{v-1}{rv}\sigma^2_{C(x)} \tag{9}$$

PROOF. The error variance of ensemble A, given some mean correlation δ of error among the classifiers in the ensemble, is

$$\sigma^2_{A(x)} = \left(\frac{1+\delta(Kv-1)}{Kv}\right)\bar{\sigma}^2_{A(x)}, \tag{10}$$

where $\bar{\sigma}^2_{A(x)}$ is the common variance of $\sigma^2_{A_i(x)}$. Mean correlation δ is given by

$$\delta = \frac{1}{(Kv)(Kv-1)}\sum_{m=1}^{Kv}\sum_{m\neq l} corr(\eta_m, \eta_l), \tag{11}$$

where $corr(\eta_m, \eta_l)$ is the correlation between the errors of classifiers A_m and A_l [Tumer and Ghosh 1996].

To simplify the computation of error correlation between A_m and A_l, we assume that if they are trained with overlapping data then $corr(\eta_m, \eta_l) = 1$, and if they are trained with disjoint data, then $corr(\eta_m, \eta_l) = 0$. Given this assumption, the correlation between A_m and A_l can be computed as follows.

$$corr(\eta_m, \eta_l) = \begin{cases} \frac{v-2}{v-1} & \text{if } A_m, A_l \in A^i \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

That is, the error correlation between classifiers A_m and A_l is nonzero only if they are in the same batch of classifiers A^i. If A_m and A_l are from the same batch, they have $v-2$ partitions of training data in common. This is because our algorithm trains each classifier using $v-1$ partitions of training data, and each batch contains v partitions in total. However, if the classifiers are not from the same batch, they do not have any common training data.

In the worst case, all v classifiers of the ith batch will remain in the ensemble A. This worst case is probably infrequent, since the ensemble updating algorithm allows some classifiers in the ith batch to be replaced while others remain in the ensemble. However, in the worst case the ensemble is updated each time by a replacement of a whole batch of v classifiers by a new batch of v classifiers. In this case, each classifier will be correlated with $v-1$ classifiers. The mean correlation therefore becomes

$$\delta \leq \frac{1}{(Kv)(Kv-1)} Kv(v-1)\frac{v-2}{v-1} = \frac{v-2}{Kv-1}. \tag{13}$$

Substituting this bound for δ into Eq. (10), and following logic similar to that used in the proof of Lemma 1, we obtain

$$\begin{aligned} \sigma^2_{A(x)} &\leq \left(\frac{1 + \frac{v-2}{Kv-1}(Kv-1)}{Kv} \right) \bar{\sigma}^2_{A(x)} \\ &= \frac{v-1}{Kv}\bar{\sigma}^2_{A(x)} = \frac{v-1}{Kv}\frac{1}{Kv}\sum_{i=1}^{Kv}\sigma^2_{A_i(x)}. \end{aligned}$$

Now first using Eq. (7) and then (8) we can deduce

$$\begin{aligned} &= \frac{v-1}{Kv}\frac{1}{Kv}\sum_{i=1}^{Kv}\left(\frac{1}{r^2}\sum_{j=i}^{r+i-1}\sigma^2_{C_j(x)}\right) \\ &= \frac{v-1}{rv}\sigma^2_{C(x)} \end{aligned} \tag{14}$$

□

For example, if $v = 2$ and $r = 2$ then we anticipate an error reduction by a factor of approximately 4.

Lemma 2 accounts for correlation between the classifiers in each ensemble, but it does not account for concept-drift in the data stream. In order to analyze error in the presence of concept-drift, we must introduce a new term for the *magnitude of drift*.

Definition 1. The magnitude of drift ρ_d is the maximum error introduced to a classifier due to concept-drift. That is, every time a new data chunk appears, the error variance of a classifier is incremented ρ_d times due to concept-drift.

For example, let D_j (with $j \in [i, i+r-1]$) be a data chunk in a window of r of consecutive data chunks $\{D_i, \ldots, D_{i+r-1}\}$, and let C_j be the classifier trained with data chunk D_j. In addition, let $\sigma^2_{C_j(x)}$ be the error variance of C_j in the absence of concept-drift. The actual error variance $\hat{\sigma}^2_{C_j(x)}$ of classifier C_j in the presence of concept-drift is then given by

$$\hat{\sigma}^2_{C_j(x)} = (1+\rho_d)^{(i+r-1)-j}\sigma^2_{C_j(x)}. \tag{15}$$

In other words, $\hat{\sigma}^2_{C_j(x)}$ is the actual error variance of the jth classifier C_j in the presence of concept-drift when the last data chunk D_{i+r-1} in the window appears. The following lemma generalizes the results of Lemma 2 to account for such concept-drift.

LEMMA 3. *Let $\hat{\sigma}^2_{A(x)}$ be the error variance of EMPC in the presence of concept-drift, let $\sigma^2_{C(x)}$ be the error variance of SPC, and let ρ_d be magnitude of drift given by Definition* 1.

Error variance $\hat{\sigma}^2_{A(x)}$ is bounded by

$$\hat{\sigma}^2_{A(x)} \leq \frac{(v-1)(1+\rho_d)^{r-1}}{rv}\sigma^2_{C(x)}. \tag{16}$$

PROOF. Replacing $\sigma^2_{C_j(x)}$ with $\hat{\sigma}^2_{C_j(x)}$ in Eq. (8) and using Eq. (15) and Lemma 2, we get

$$\begin{aligned}
\hat{\sigma}^2_{A(x)} &\leq \frac{v-1}{K^2v^2}\sum_{i=1}^{Kv}\frac{1}{r^2}\sum_{j=i}^{r+i-1}\hat{\sigma}^2_{C_j(x)} \\
&= \frac{v-1}{K^2r^2v^2}\sum_{i=1}^{Kv}\sum_{j=i}^{r+i-1}(1+\rho_d)^{(i+r-1)-j}\sigma^2_{C_j(x)} \\
&\leq \frac{v-1}{K^2r^2v^2}\sum_{i=1}^{Kv}(1+\rho_d)^{r-1}\sum_{j=i}^{r+i-1}\sigma^2_{C_j(x)} \\
&= \frac{(v-1)(1+\rho_d)^{r-1}}{K^2r^2v^2}\sum_{i=1}^{Kv}\sum_{j=i}^{r+i-1}\sigma^2_{C_j(x)} \\
&= \frac{(v-1)(1+\rho_d)^{r-1}}{Krv}\bar{\sigma}^2_{C(x)} \\
&= \frac{(v-1)(1+\rho_d)^{r-1}}{rv}\sigma^2_{C(x)}.
\end{aligned} \tag{17}$$

□

Therefore, we expect a reduction of error provided that

$$E_R = \frac{(v-1)(1+\rho_d)^{r-1}}{rv} \leq 1. \tag{18}$$

That is, the ratio E_R of EMPC error to SPC error in the presence of concept-drift must be no greater than 1. As we increase r and v, the relative error therefore decreases up to a certain threshold, after which it flattens or increases. We next empirically seek ideal values of r and v for reducing error in the presence of concept-drift.

3.3. Empirical Error Reduction

For a given partition size v, increasing the window size r only yields reduced error up to a certain point. After that, increasing r actually hurts the performance of our algorithm, because inequality (18) is violated. The upper bound of r depends on the magnitude of drift ρ_d.

Figure 2 shows the relative error E_R for $v = 2$, and different values of ρ_d, for increasing r. It is clear from the graph that for lower values of ρ_d, increasing r reduces the relative error by a greater margin. However, in all cases after r exceeds a certain threshold, E_R becomes greater than one.

Although it may not be possible to know the actual value of ρ_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 work better, and vice versa. However, the best performance-cost trade-off is found for $r = 2$ or $r = 3$. We have used $r = 2$ in our experiments. Similarly, the upper bound of v can be derived from inequality (18) for a fixed value of r. It should be noted that if v is increased, running time also increases. From our experiments, we obtained the best performance-cost trade-off for $v = 5$.

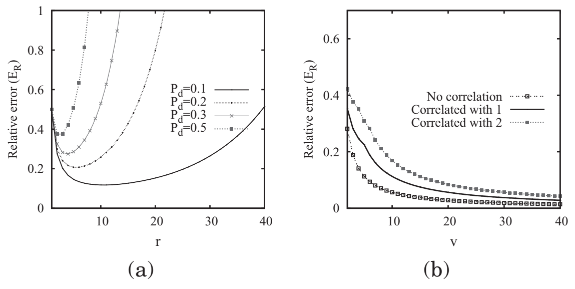


Fig. 2. Error reduction by increasing r and v.

3.4. Time Complexity of EMPC

The time complexity of the algorithm is $O(vn(Ks + f(rs)))$, where n is the total number of data chunks, s is the size of each chunk, and $f(z)$ is the time required to build a classifier on a training data of size z. Since v is constant, the complexity becomes $O(n(Ks + f(rs)))$. This is at most a constant factor rv slower than the closest related work [Wang et al. 2003], but with the advantage of significantly reduced error.

4. MALICIOUS CODE DETECTION

Malware is a major source of cyber attacks. Some malware varieties are purely static; each instance is an exact copy of the instance that propagated it. These are relatively easy to detect and filter once a single instance has been identified. However, a much more significant body of current-day malware is polymorphic. Polymorphic malware self-modifies during propagation so that each instance has a unique syntax but carries a semantically identical malicious payload.

The antivirus community invests significant effort and manpower toward devising, automating, and deploying algorithms that detect particular malware instances and polymorphic malware families that have been identified and analyzed by human experts. This has led to an escalating arms race between malware authors and antiviral defenders, in which each camp seeks to develop offenses and defenses that counter the recent advances of the other. With the increasing ease of malware development and the exponential growth of malware variants, many believe that this race will ultimately prove to be a losing battle for the defenders.

The malicious code detection problem can be modeled as a data mining problem for a stream having both infinite length and concept-drift. Concept-drift occurs as polymorphic malware mutates, and as attackers and defenders introduce new technologies to the arms race. This conceptualization invites application of our stream classification technique to automate the detection of new malicious executables.

Feature extraction using n-gram analysis involves extracting all possible n-grams from the given dataset (training set), and selecting the best n-grams among them. Each such n-gram is a feature. That is, an n-gram is a sequence of n bytes. Before extracting n-grams, we preprocess the binary executables by converting them to hexdump files. Here, the granularity level is one byte. We apply the UNIX `hexdump` utility to convert the binary executable files into text files (*hexdump files*) containing the hexadecimal numbers corresponding to each byte of the binary. This process is performed to ensure safe and easy portability of the binary executables. In a nondistributed framework, the feature extraction process consists of two phases: feature extraction and feature selection, described shortly. Our cloud computing variant of this traditional technique is presented in Section 4.2.

4.1. Nondistributed Feature Extraction and Selection

In a nondistributed setting, feature extraction proceeds as follows. Each hexdump file is scanned by sliding an n-byte window over its content. Each n-byte sequence that appears in the window is an n-gram. For each n-gram g, we tally the total number t_g of file instances in which g appears, as well as the total number $p_g \leq t_g$ of these that are positive (i.e., malicious executables).

This involves maintaining a hash table T of all n-grams encountered so far. If g is not found in T, then g is added to T with counts $t_g = 1$ and $p_g \in \{0, 1\}$ depending on whether the current file has a negative or positive class label. If g is already in T, then t_g is incremented and p_g is conditionally incremented depending on the file's label. When all hexdump files have been scanned, T contains all the unique n-grams in the dataset along with their frequencies in the positive instances and in total.

It is not always practical to use all n-gram features extracted from all the files corresponding to the current chunk. The exponential number of such n-grams may introduce unacceptable memory overhead, slow the training process, or confuse the classifier with large numbers of noisy, redundant, or irrelevant features. To avoid these pitfalls, candidate n-gram features must be sorted according to a selection criterion so that only the best ones are selected.

We choose *information gain* as the selection criterion, because it is one of the most effective criteria used in literature for selecting the best features. Information gain can be defined as a measure of the effectiveness of an attribute (i.e., feature) for classifying the training data. If we split the training data based on the values of this attribute, then information gain measures the expected reduction in entropy after the split. The more an attribute reduces entropy in the training data, the better that attribute is for classifying the data.

Given an instance set I with positive instances $P \subseteq I$, and an attribute $R : I \to V$ that maps instances to values, the information gain is given by

$$G(R) = H(p, t) - \sum_{v \in V} \frac{|I_{Rv}|}{t} H\big(|I_{Rv} \cap P|, |I_{Rv}|\big), \tag{19}$$

where $p = |P|$ is the total number of positive instances, $t = |I|$ is the total size of the instance set, $I_{Rv} = \{i \in I \mid R(i) = v\}$ is the set of instances in which attribute R has value v, and entropy H is given by the formula

$$H(x, y) = -\frac{x}{y} \log_2 \left(\frac{x}{y}\right) - \frac{y - x}{y} \log_2 \left(\frac{y - x}{y}\right). \tag{20}$$

In the context of malware detection, we consider each n-gram g to be an attribute $R_g : I \to \{0, 1\}$ that maps each instance $i \in I$ to 1 if g is present in i and to 0 otherwise. Thus, $|I_{R_g 1}| = t_g$ is the total number of instances that contain g, and $|I_{R_g 0}| = t - t_g$ is the total number that do not. Likewise, $|I_{R_g 1} \cap P| = p_g$ is the total number of positive instances that contain g, and $|I_{R_g 0} \cap P| = p - p_g$ is the total number of positive instances that do not. Substituting these formulas into Eqs. (19) and (20) we obtain

$$G(R_g) = H(p, t) - \frac{t_g}{t} H(p_g, t_g) - \frac{t - t_g}{t} H(p - p_g, t - t_g).$$

We henceforth write

$$\hat{G}(p_g, t_g, p, t) = G(R_g). \tag{21}$$

The feature set can therefore be pruned by selecting the S features with greatest information gain $G(R_g)$ based on the preceding. This can be efficiently accomplished with a min-heap data structure of size S that stores the S best features seen so far, keyed by information gain. As new features are considered, their information gains are compared against the heap's root. If the gain of the new feature is greater than that of the root, the root is discarded and the new feature inserted into the heap. Otherwise the new feature is discarded and feature selection continues.

4.2. Distributed Feature Extraction and Selection

There are several drawbacks related to the nondistributed feature extraction and selection approach just described.

—The total number of extracted *n*-gram features might be very large. For example, the total number of 4-grams in one chunk is around 200 million. It might not be possible to store all of them in main memory. One obvious solution is to store the *n*-grams in a disk file, but this introduces unacceptable overhead due to the cost of disk read/write operations.
—If colliding features in hash table T are not sorted, then a linear search is required for each scanned *n*-gram during feature extraction to test whether it is already in T. If they are sorted, then the linear search is required during insertion. In either case, the time to extract all *n*-grams is worst case quadratic in the total number N of *n*-grams in each chunk, an impractical amount of time when $N \approx 10^8$.
—Similarly, the nondistributed feature selection process requires a sort of the *n*-grams in each chunk. In general, this requires $O(N \log N)$ time, which is impractical when N is large.

In order to efficiently and effectively tackle the drawbacks of the nondistributed feature extraction and selection approach, we leverage the power of cloud computing. This allows feature extraction, *n*-gram sorting, and feature selection to be performed in parallel, utilizing the Hadoop *MapReduce* framework.

MapReduce [Dean and Ghemawat 2008] is an increasingly popular distributed programming paradigm used in cloud computing environments. The model processes large datasets in parallel, distributing the workload across many nodes (machines) in a share-nothing fashion. The main focus is to simplify the processing of large datasets using inexpensive cluster computers. Another objective is ease of usability with both load balancing and fault tolerance.

MapReduce is named for its two primary functions. The *Map* function breaks jobs down into subtasks to be distributed to available nodes, whereas its dual, *Reduce*, aggregates the results of completed subtasks. We will henceforth refer to nodes performing these functions as *mappers* and *reducers*, respectively. The details of the MapReduce process for *n*-gram feature extraction and selection are explained in the Appendix. In this section, we give a high-level overview of the approach.

Each training chunk containing N training files are used to extract the *n*-grams. These training files are first distributed among m nodes (machines) by the Hadoop Distributed File System (HDFS) (Figure 3, step 1). Quantity m is selected by HDFS depending on system availability. Each node then independently extracts *n*-grams from the subset of training files supplied to the node using the technique discussed in Section 4.1 (Figure 3, step 2). When all nodes finish their jobs, the *n*-grams extracted from each node are collated (Figure 3, step 3).

For example, suppose Node 1 observes *n*-gram *abc* in one positive instance (i.e., a malicious training file) while Node 2 observes it in a negative (i.e., benign) instance. This is denoted by pairs $\langle abc, +\rangle$ and $\langle abc, -\rangle$ under Nodes 1 and 2 (respectively) in Figure 3. When the *n*-grams are combined, the labels of instances containing identical *n*-grams are aggregated. Therefore, the aggregated pair for *abc* is $\langle abc, +-\rangle$.

The combined *n*-grams are distributed to q reducers (with q chosen by HDFS based on system availability). Each reducer first tallies the aggregated labels to obtain a

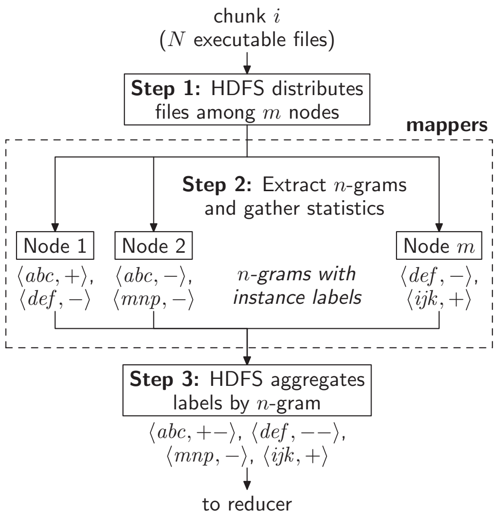

chunk i
(N executable files)
Step 1: HDFS distributes files among m nodes
mappers
Step 2: Extract n-grams and gather statistics
Node 1
Node 2
Node m
$\langle abc,+\rangle$, $\langle def,-\rangle$
$\langle abc,-\rangle$, $\langle mnp,-\rangle$
n-grams with instance labels
$\langle def,-\rangle$, $\langle ijk,+\rangle$
Step 3: HDFS aggregates labels by n-gram
$\langle abc,+-\rangle$, $\langle def,--\rangle$, $\langle mnp,-\rangle$, $\langle ijk,+\rangle$
to reducer

Fig. 3. Feature extraction with Hadoop Map.

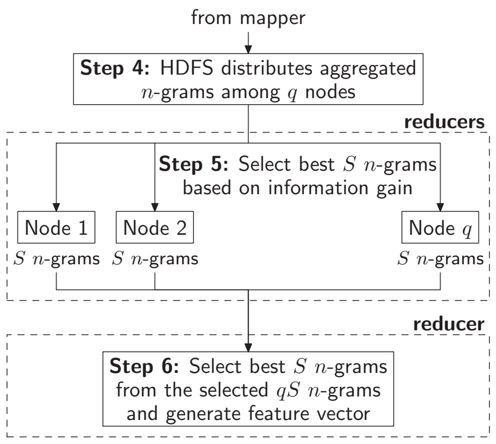

from mapper
Step 4: HDFS distributes aggregated n-grams among q nodes
reducers
Step 5: Select best S n-grams based on information gain
Node 1
Node 2
Node q
S n-grams
S n-grams
S n-grams
reducer
Step 6: Select best S n-grams from the selected qS n-grams and generate feature vector

Fig. 4. Feature selection with Hadoop Reduce.

positive count and a total count. In the case of n-gram abc, we obtain tallies of $p_{abc} = 1$ and $t_{abc} = 2$. The reducer uses these tallies to choose the best S n-grams (based on Eq. (21)) from the subset of n-grams supplied to the node (Figure 4, step 5). This can be done efficiently using a min-heap of size S; the process requires $O(W \log S)$ time, where W is the total number of n-grams supplied to each reducer. In contrast, the nondistributed version requires $O(W \log W)$ time. Thus, from the q reducer nodes, we obtain qS n-grams.

From these, we again select the best S by running another round of the MapReduce cycle in which the Map phase does nothing but the Reduce phase performs feature selection using only one node (Figure 4, step 6). Each feature in a feature set is binary; its value is 1 if it is present in a given instance (i.e., executable) and 0 otherwise. For each training or testing instance, we compute the feature vector whose bits consist of the feature values of the corresponding feature set. These feature vectors are used by the classifiers for training and testing.

5. EXPERIMENTS

We evaluated our approach on synthetic data, botnet traffic generated in a controlled environment, and a malware dataset. The results of the experiments are compared with several baseline methods.

5.1. Datasets

5.1.1. Synthetic Dataset. To generate synthetic data with a drifting concept, we use a moving hyperplane, given by $\sum_{i=1}^{d} a_i x_i = a_0$ [Wang et al. 2003]. 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 $\{x_1, \ldots, x_d\}$, where $x_i \in [0, 1]$. Weights $\{a_1, \ldots, a_d\}$ 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 percent of the examples, where $p = 5$ in our experiments.

There are several parameters that simulate concept-drift. We use parameters identical to those in [Wang et al. 2003]. In total, we generate 250,000 records and four different datasets having chunk sizes 250, 500, 750, and 1000, respectively. Each dataset has 50% positive instances and 50% negative.

5.1.2. Botnet Dataset. Botnets are networks of compromised hosts known as *bots*, all under the control of a human attacker known as the *botmaster* [Barford and Yegneswaran 2006]. The botmaster can issue commands to the bots to perform malicious actions, such as launching DDoS attacks, spamming, spying, and so on. Botnets are widely regarded as an enormous emerging threat to the internet community. Many cutting-edge botnets apply Peer-to-Peer (P2P) technology to reliably and covertly communicate as the botnet topology evolves. These botnets are distributed and small, making them more difficult to detect and destroy. Examples of P2P bots include Nugache [Lemos 2006], Sinit [Stewart 2003], and Trojan.Peacomm [Grizzard et al. 2007].

Botnet traffic can be viewed as a data stream having both infinite length and concept-drift. Concept-drift occurs as the bot undertakes new malicious missions or adopts differing communication strategies in response to new botmaster instructions. We therefore consider our stream classification technique to be well suited to detecting P2P botnet traffic.

We generate real P2P botnet traffic in a controlled environment using the Nugache P2P bot [Lemos 2006]. The details of the feature extraction process are discussed in Masud et al. [2008b]. There are 81 continuous attributes in total. The whole dataset consists of 30,000 records, representing one week's worth of network traffic. We generate four different datasets having chunk sizes of 30 minutes, 60 minutes, 90 minutes, and 120 minutes, respectively. Each dataset has 25% positive (botnet traffic) instances and 75% negative (benign traffic).

5.1.3. Malware Dataset. We extract a total of 38,694 benign executables from different Windows machines, and a total of 66,694 malicious executables collected from an online malware repository [VX Heavens 2010], which contains a large collection of malicious executables (viruses, worms, trojans, and back-doors). The benign executables include various applications found at the Windows installation folder, as well as other executables in the default program installation directory.

We select only the Win32 Portable Executables (PE) in both cases. Experiments with the ELF executables are a potential direction of future work. The collected 105,388 files (benign and malicious) form a data stream of 130 chunks, each consisting of 2000 instances (executable files). The stream order was chosen by sorting the malware by version and discovery date, simulating the evolving nature of Internet malware. Each chunk has 1500 benign executables (75% negative) and 500 malicious executables (25% positive). The feature extraction and selection process for this dataset is described in Sections 4.1–4.2.

Note that all these datasets are dynamic in nature. Their unbounded (potentially infinite-length) size puts them beyond the scope of purely static classification frameworks. The synthetic data also exhibits concept-drift. Although it is not possible to accurately determine whether the real datasets have concept-drift, theoretically the stream of executables should exhibit concept-drift when observed over a long period of time (see Section 1). The malware data exhibits feature evolution as evidenced by the differing set of distinguishing features identified for each chunk.

5.2. Baseline Methods

For classification, we use the Weka machine learning open-source package [Hall et al. 2009]. We apply two different classifiers: J48 decision tree and Ripper. We then compare each of the following baseline techniques to our EMPC algorithm.

BestK. This is a single-partition, single-chunk (SPC) ensemble approach, where an ensemble of the best K classifiers is used. The ensemble is created by storing all the classifiers seen so far, and selecting the best K based on expected error on the most recent training chunk. An instance is tested using simple majority voting.

Last. In this case, we only keep the classifier trained on the most recent training chunk. This can be considered an SPC approach with $K = 1$.

AWE. This is the SPC method implemented using Accuracy-Weighted classifier Ensembles [Wang et al. 2003]. It builds an ensemble of K models, where each model is trained from one data chunk. The ensemble is updated as follows. Let C_n be the classifier built on the most recent training chunk. From the existing K models and the newest model C_n, the K best models are selected based on their error on the most recent training chunk. Selection is based on weighted voting where the weight of each model is inversely proportional to the error of the model on the most recent training chunk.

All. This SPC uses an ensemble of all the classifiers seen so far. The new data chunk is tested with this ensemble by simple voting among the classifiers. Since this is an SPC approach, each classifier is trained from only one data chunk.

As mentioned in Section 3.3, we obtain the optimal values of r and v to be between 2 and 3, and between 3 and 5, respectively, for most datasets. Unless mentioned otherwise, we use $r = 2$ and $v = 5$ in our experiments. To obtain a fair comparison, we use the same value for K (ensemble size) in EMPC and all baseline techniques.

5.3. Hadoop Distributed System Setup

The distributed system on which we performed our experiments consists of a cluster of ten nodes. Each node has the same hardware configuration: an Intel Pentium IV 2.8 GHz processor, 4GB main memory, and 640GB hard disk space. The software environment consists of a Ubuntu 9.10 operating system, the Hadoop-0.20.1 distributed computing platform, the JDK 1.6 Java development platform, and a 100MB LAN network link.

5.4. Performance Study

In this section we compare the results of all five techniques: EMPC, AWE, *BestK*, *All*, and *Last*. As each new data chunk appears, we test each ensemble/classifier on the

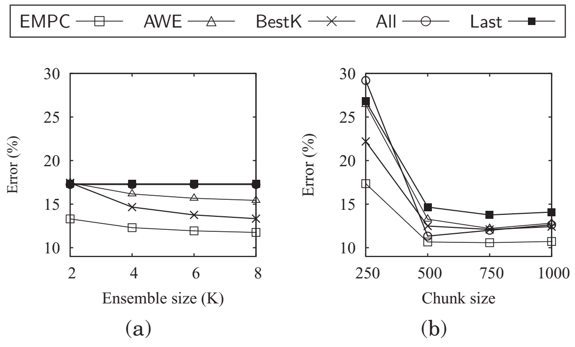


Fig. 5. Error rates for synthetic data.

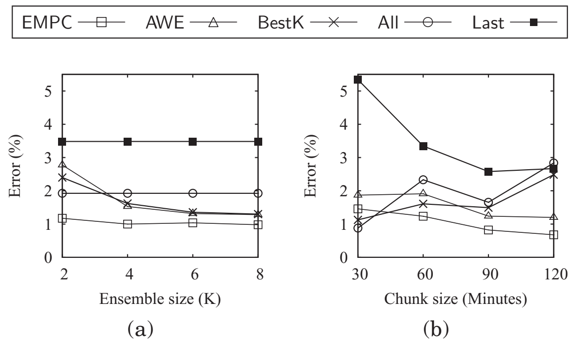


Fig. 6. Error rates for botnet data.

new data and update its accuracy, false positive rate, and false negative rate. In all the results shown here we fix the parameter values of $v = 5$ and $r = 2$, and the base learner is a decision tree unless stated otherwise.

Figure 5(a) shows the error rates for different ensemble sizes K, averaged over four different chunk sizes on synthetic data. It is evident that EMPC has the lowest error among all approaches. The accuracy does not improve much after $K = 8$. Methods AWE and *BestK* also show similar characteristics. Methods *All* and *Last* do not depend on K, so their error rates remain the same for any K.

Figure 5(b) shows the error rates for four different chunk sizes of each method averaged over different ensemble sizes K on synthetic data. Again, EMPC has the lowest error of all. The error of EMPC is also lower for larger chunk sizes, since these provide more training data for each classifier.

Figure 6(a) shows the error rates for botnet data over different ensemble sizes K averaged over four different chunk sizes. Figure 6(b) shows the error rates for the same data over four different chunk sizes averaged over the different ensemble sizes K. In all cases, EMPC has the lowest error rate among all approaches.

Figure 7(a) shows the error rates for different ensemble sizes K on malware data. EMPC outperforms all other methods and reaches the lowest error rate when $K = 3$. Figure 7(b) shows the error rates for the same data over different feature set sizes. EMPC outperforms all other methods and reaches the lowest error rate when the feature size is 2000.

Tables (a) and (b) of Table I report the error rates for decision tree and Ripper learning algorithms, respectively, on synthetic data for different ensemble sizes K and

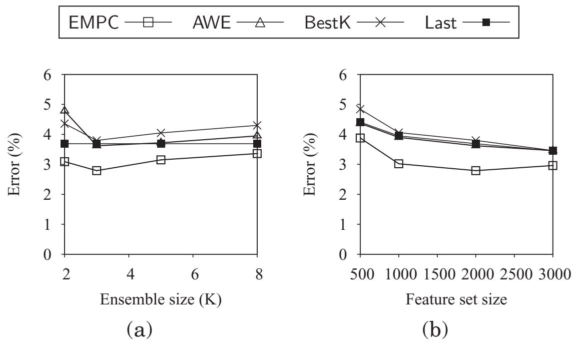


Fig. 7. Error rates for malware data.

Table I. Error Rates on Synthetic Data

(a) Decision tree

Chunk size	$K = 2$			$K = 4$			$K = 8$			*All*	*Last*
	EMPC	AWE	*BestK*	EMPC	AWE	*BestK*	EMPC	AWE	*BestK*		
250	**19.3**	26.8	26.9	**17.3**	26.5	22.1	**16.2**	26.1	19.5	29.2	26.8
500	**11.4**	14.8	14.7	**10.6**	13.2	12.4	**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**	11.3	11.2	15.8	13.8
1000	**11.4**	14.3	14.3	**10.7**	12.8	12.2	**10.3**	11.9	11.4	12.6	14.1

(b) Ripper

Chunk size	$K = 2$			$K = 6$			$K = 8$			*All*	*Last*
	EMPC	AWE	*BestK*	EMPC	AWE	*BestK*	EMPC	AWE	*BestK*		
250	**19.2**	26.5	26.0	**17.6**	26.2	22.4	**16.8**	25.9	20.9	30.4	26.3
500	**11.5**	14.2	13.9	**10.8**	13.0	12.3	**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**	11.5	11.5	15.7	13.3
1000	**11.1**	13.8	13.7	**10.6**	12.5	12.3	**10.2**	11.9	11.8	12.6	13.6

Table II. Error Rates on Malware Data Using Decision Tree

Feature set size	$K = 3$			$K = 5$			$K = 8$			*Last*
	EMPC	AWE	*BestK*	EMPC	AWE	*BestK*	EMPC	AWE	*BestK*	
500	**3.88**	4.37	4.84	**3.96**	4.40	5.07	**4.08**	4.35	5.36	4.41
1000	**3.02**	3.90	4.06	**3.09**	4.05	4.35	**3.21**	4.34	4.95	3.95
2000	**2.79**	3.62	3.80	**3.15**	3.72	4.05	**3.36**	3.95	4.30	3.69
3000	**2.96**	3.46	3.46	**3.20**	3.41	3.34	**3.31**	3.58	3.64	3.45

chunk sizes. In all tables, we see that EMPC has the lowest error rate for all ensemble sizes (shown in bold).

Table II reports the error rates for malware data over different ensemble sizes K and feature set sizes. Once again, EMPC has the lowest overall error rate for all values of K and feature set size (shown in bold).

Figure 8 shows the sensitivity of parameter r on the EMPC error rates and runtimes over synthetic data. Figure 8(a) shows the error rates over different values of r for fixed parameters $v = 5$ and $K = 8$. The highest reduction in error rate occurs when r is increased from 1 to 2. Note that $r = 1$ means single-chunk training. We observe no significant reduction in error rate for higher values of r, which follows from our analysis of parameter r on concept-drifting data in Section 3.3. However, the runtime keeps increasing, as shown in Figure 8(b). The best trade-off between runtime and error therefore occurs for $r = 2$.

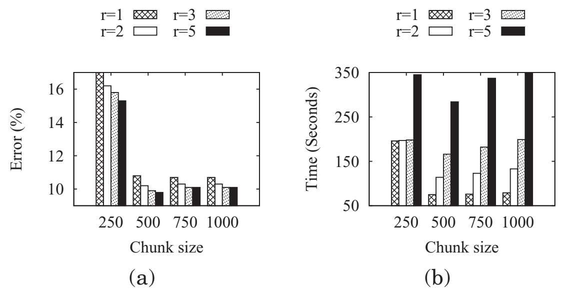


Fig. 8. Sensitivity of r on error rate and runtime.

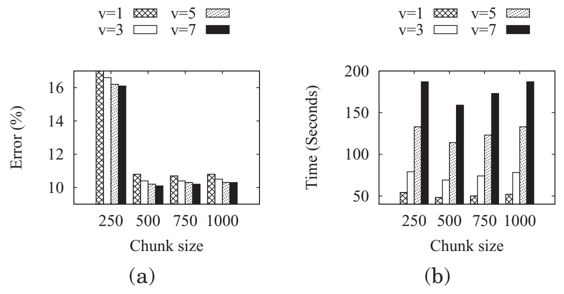


Fig. 9. Sensitivity of v on error rate and runtime.

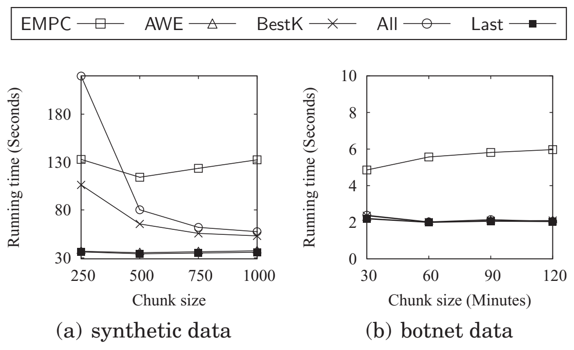
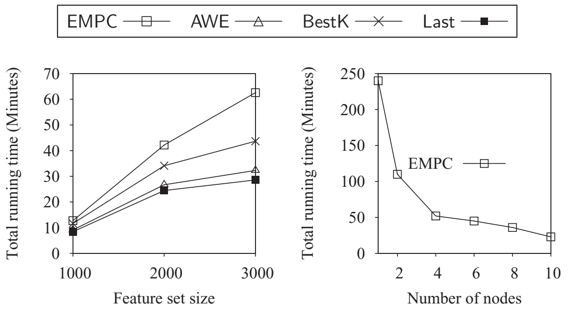


Fig. 10. Total runtimes.

Figure 9 shows a similar trend for parameter v. Note that $v = 1$ (i.e., a single partition ensemble) is the base case, and $v > 1$ is the multipartition ensemble approach. We observe no significant improvement after $v = 5$, although the runtime keeps increasing. This result is also consistent with our analysis of the upper bounds of v, explained in Section 3.3. We choose $v = 5$ as the best trade-off between time and error.

Figure 10(a) shows the total running times of different methods on synthetic data for $K = 8$, $v = 5$ and $r = 2$. Note that the runtime of EMPC is within 5 times of that of AWE. This supports our complexity analysis in Section 3.4, which concludes that the runtime of EMPC is at most rv times that of AWE. The runtimes of EMPC on botnet data shown in Figure 10(b) have similar characteristics. All runtimes shown in Figure 10 include both training and testing time. Although the total training time of EMPC is higher than that of AWE, the total testing times are almost the same for



(a) training and testing (b) feature extraction and selection

Fig. 11. Runtimes for malware data.

Table III. Error Comparison for Fixed Ensemble Size

Chunk size	J48 EMPC ($K=2$)	J48 AWE ($K=10$)	Ripper EMPC ($K=2$)	Ripper AWE ($K=10$)
250	**19.9**	26.1	**21.0**	26.1
500	**11.7**	12.5	**12.2**	12.6
1000	**11.4**	12.5	**11.8**	13.0

both techniques. Considering that training can be done offline, we conclude that both these techniques have comparable runtime performances in classifying data streams. However, EMPC affords users the additional flexibility of choosing between better performance or shorter training times by varying parameters r and v.

Figure 11(a) shows the total training and testing runtimes of each method (excluding the extraction time) on the malware dataset for $K = 3$, $v = 5$, and $r = 2$. Although the total training time of EMPC is higher than other techniques, the total testing times do not differ that much across all techniques. Once again, the user may tune v and r to trade lower accuracy for faster runtimes if necessary, but only up to a point. Increasing v and r beyond their optimal values does not yield any further accuracy improvement. For example, in this experiment, we observed that with $v = 5$ and $r = 2$ we received the best results.

Figure 11(b) shows the total feature extraction and selection runtimes for the cloud-based, distributed approach discussed in Section 4.2. When the number of nodes is 1, the results are for the single-machine (nondistributed) approach discussed in Section 4.1. It is evident that the time is linearly decreasing when we utilize more machines (nodes) in the cloud. For example, when 2 and 6 nodes are used, feature extraction and selection time are 110 minutes and 45 minutes, respectively. Therefore, utilizing ten nodes or more dramatically improves the running time.

In Table III we also report the results of using equal numbers of classifiers in EMPC and AWE by setting $K = 10$ in AWE, and $K = 2$, $v = 5$, and $r = 1$ in EMPC. We observe that the error rate of EMPC is lower than that of AWE in all chunk sizes. For example, using decision trees with chunk size 250, EMPC's error rate is 19.9%, whereas that of AWE is 26.1%.

Two important conclusions follow from this result. First, we see that merely increasing the ensemble size of AWE by a factor of v (making it equal to Kv) does not suffice to reduce its error rate to that of EMPC. Second, even if we use the same training set size for both methods (i.e., $r = 1$), EMPC's error rate still remains lower.

6. DISCUSSION

Our work considers a feature space consisting of purely syntactic features: binary n-grams drawn from executable code segments, static data segments, headers, and all other content of untrusted files. Higher-level structural features such as call- and control-flow graphs, and dynamic features such as runtime traces, are beyond our current scope. Nevertheless, n-gram features have been observed to have very high discriminatory power for malware detection, as demonstrated by a large body of prior work (see Section 2) as well as our experiments (see Section 5.4). This is in part because n-gram sets that span the entire binary file content, including headers and data tables, capture important low-level structural details that are often abstracted away by higher-level representations. For example, malware often contains hand-written assembly code that has been assembled and linked using nonstandard tools. This allows attackers to implement binary obfuscations and low-level exploits not available from higher-level source languages and standard compilers. As a result, malware often contains unusual instruction encodings, header structures, and link tables whose abnormalities can only be seen at the raw binary level, not in assembly code listings, control-flow graphs, or system API call traces. Expanding the feature space to include these additional higher-level features requires an efficient and reliable method of harvesting them and assessing their relative discriminatory power during feature selection, and is reserved as a subject of future work.

The empirical results reported in Section 5.4 confirm the analysis presented in Section 3 that shows that multipartition, multichunk approaches should perform better than single-chunk, single-partition approaches. Intuitively, a classifier trained on multiple chunks should have better prediction accuracy than a classifier trained on a single chunk because of the larger training data. Furthermore, if more than one classifier is trained by multipartitioning the training data, the prediction accuracy of the resulting ensemble of classifiers should be higher than a single classifier trained from the same training data because of the error reduction power of an ensemble over single classifier. In addition, the accuracy advantages of EMPC can be traced to two important differences between our work and that of AWE. First, when a classifier is removed during ensemble updating in AWE, all information obtained from the corresponding chunk is forgotten; but in EMPC, one or more classifiers from an earlier chunk may survive. Thus, EMPC ensemble updating tends to retain more information than that of AWE, leading to a better ensemble. Second, AWE requires at least Kv data chunks, whereas EMPC requires at least $K + r - 1$ data chunks to obtain Kv classifiers. Thus, AWE tends to keep much older classifiers in the ensemble than EMPC, leading to some outdated classifiers that can have a negative effect on the classification accuracy.

However, the higher accuracy comes with an increased cost in running time. Theoretically, EMPC is at most rv times slower than AWE, its closest competitor in accuracy (Section 3.4). This is also evident in the empirical evaluation (Section 5.4), which shows that the running time of EMPC is within 5 times that of AWE (for $r = 2$ and $v = 5$). However, some optimizations can be adopted to reduce the runtime cost. First, parallelization of training for each partition can be easily implemented, reducing the training time by a factor of v. Second, classification by each model in the ensemble can also be done in parallel, thereby reducing the classification time by a factor of Kv. Therefore, parallelization of training and classification should reduce the running time at least by a factor of v, making the runtime close to that of AWE. Alternatively, if parallelization is not available, parameters v and r can be lowered to sacrifice prediction accuracy for lower runtime cost. In this case, the desired balance between runtime and prediction accuracy can be obtained by evaluating the first few chunks of the stream with different values of v and r and choosing the most suitable values.

7. CONCLUSION

Many intrusion detection problems can be formulated as classification problems for infinite-length, concept-drifting data streams. Concept-drift occurs in these streams as attackers react and adapt to defenses. We formulated both malicious code detection and botnet traffic detection as such problems, and introduced EMPC, a novel ensemble learning technique for automated classification of infinite-length, concept-drifting streams. Applying EMPC to real data streams obtained from polymorphic malware and botnet traffic samples yielded better detection accuracies than other stream data classification techniques. This shows that the approach is useful and effective for both intrusion detection and more general data stream classification.

EMPC uses generalized, multipartition, multichunk ensemble learning. Both theoretical and empirical evaluation of the technique show that it significantly reduces the expected classification error over existing single-partition, single-chunk ensemble methods. Moreover, we show that EMPC can be elegantly implemented in a cloud computing framework based on MapReduce [Dean and Ghemawat 2008]. The result is a low-cost, scalable stream classification framework with high classification accuracy and low runtime overhead.

At least two extensions to our technique offer promising directions of future work. First, our current feature selection procedure limits its attention to the best S features based on information gain as the selection criterion. The classification accuracy could potentially be improved by leveraging recent work on supervised dimensionality reduction techniques [Rish et al. 2008; Sajama and Orlitsky 2005] for improved feature selection. Second, the runtime performance of our approach could be improved by exploiting additional parallelism available in the cloud computing architecture. For example, the classifiers of an ensemble could be run in parallel as mappers in a MapReduce framework, with reducers that aggregate the results for voting. Similarly, the candidate classifiers for the next ensemble could be trained and evaluated in parallel. Reformulating the ensemble components of the system in this way could lead to significantly shortened processing times, and hence opportunities to devote more processing time to classification for improved accuracy.

APPENDIX

We used the open-source Hadoop MapReduce framework by Apache [2010] to implement our experiments. We here provide some of the algorithmic details of the Hadoop MapReduce feature extraction and selection algorithm described at a high level in Section 4.

The *Map* function in a MapReduce framework takes a key-value pair as input and yields a list of intermediate key-value pairs for each.

$$Map : (MKey \times MVal) \rightarrow (RKey \times RVal)^*$$

All the *Map* tasks are processed in parallel by each node in the cluster without sharing data with other nodes. Hadoop collates the output of the *Map* tasks by grouping each set of intermediate values $V \subseteq RVal$ that share a common intermediate key $k \in RKey$. The resulting collated pairs (k, V) are then streamed to *Reduce* nodes. Each reducer in a Hadoop MapReduce framework therefore receives a list of multiple (k, V) pairs, issued by Hadoop one at a time in an iterative fashion. *Reduce* can therefore be understood as a function having signature

$$Reduce : (RKey \times RVal^*)^* \rightarrow Val.$$

Codomain *Val* is the type of the final results of the MapReduce cycle.

In our framework, *Map* keys (*MKey*) are binary file identifiers (e.g., filenames), and *Map* values (*MVal*) are the file contents in bytes. *Reduce* keys (*RKey*) are *n*-gram features, and their corresponding values (*RVal*) are the class labels of the file instances whence they were found. Algorithm 2 shows the feature extraction procedure that *Map* nodes use to map the former to the latter.

ALGORITHM 2: *Map*(*file_id*, *bytes*)

```
Input: file $file\_id$ with content $bytes$
Output: list of pairs $(g, l)$, where $g$ is an $n$-gram and $l$ is $file\_id$'s label
1: $T \leftarrow \emptyset$
2: for all $n$-grams $g$ in $bytes$ do
3:     $T \leftarrow T \cup \{(g, labelof(file\_id))\}$
4: end for
5: for all $(g, l) \in T$ do
6:     print $(g, l)$
7: end for
```

ALGORITHM 3: $Reduce_{p,t}(F)$

```
Input: list $F$ of $(g, L)$ pairs, where $g$ is an $n$-gram and $L$ is a list of class labels; total size $t$ of
    original instance set; total number $p$ of positive instances
Output: $S$ pairs $(g, i)$, where $i$ is the information gain of $n$-gram $g$
1: heap $h$ /* empty min-heap */
2: for all $(g, L)$ in $F$ do
3:     $t' \leftarrow 0$
4:     $p' \leftarrow 0$
5:     for all $l$ in $L$ do
6:         $t' \leftarrow t' + 1$
7:         if $l = +$ then
8:             $p' \leftarrow p' + 1$
9:         end if
10:    end for
11:    $i \leftarrow \hat{G}(p', t', p, t)$ /* see Equation 21 */
12:    if $h.size < S$ then
13:        $h.insert(i_{\langle g \rangle})$
14:    else if $(h.root < i)$ then
15:        $h.replace(h.root, i_{\langle g \rangle})$
16:    end if
17: end for
18: for all $i_{\langle g \rangle}$ in $h$ do
19:     print $(g, i)$
20: end for
```

Lines 5–10 of Algorithm 3 tally the class labels reported by *Map* to obtain positive and negative instance counts for each *n*-gram. These form a basis for computing the information gain of each *n*-gram in line 11, as described in Section 4. Lines 12–16 use a min-heap data structure h to filter all but the best S features as evaluated by information gain. The final best S features encountered are returned by lines 18–20.

The q reducers in the Hadoop system therefore yield a total of qS candidate features and their information gains. These are streamed to a second reducer that simply implements the last half of Algorithm 3 to select the best S features.

REFERENCES

Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. 2006. A framework for on-demand classification of evolving data streams. *IEEE Trans. Knowl. Data Engin. 18,* 5, 577–589.

Aha, D. W., Kibler, D., and Albert, M. K. 1991. Instance-based learning algorithms. *Mach. Learn. 6*, 37–66.

Apache. 2010. Hadoop. hadoop.apache.org.

Barford, P. and Yegneswaran, V. 2006. An inside look at botnets. In *Malware Detection*, Advances in Information Security, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds., Springer, 171–192.

Bifet, A., Holmes, G., Pfahringer, B., Kirkby, R., and Gavaldà, R. 2009. New ensemble methods for evolving data streams. In *Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 139–148.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. 1992. A training algorithm for optimal margin classifiers. In *Proceedings of the 5th ACM Workshop on Computational Learning Theory*. 144–152.

Chen, S., Wang, H., Zhou, S., and Yu, P. S. 2008. Stop chasing trends: Discovering high order models in evolving data. In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE)*. 923–932.

Cohen, W. W. 1996. Learning rules that classify e-mail. In *Proceedings of the AAAI Spring Symposium on Machine Learning in Information Access*. 18–25.

Computer Economics, Inc. 2007. Malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code. http://www.computereconomics.com/article.cfm?id=1225.

Crandall, J. R., Su, Z., Wu, S. F., and Chong, F. T. 2005. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 235–248.

Dean, J. and Ghemawat, S. 2008. MapReduce: Simplified data processing on large clusters. *Comm. ACM 51,* 1, 107–113.

Domingos, P. and Hulten, G. 2000. Mining high-speed data streams. In *Proceedings of the 6th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 71–80.

Fan, W. 2004. Systematic data selection to mine concept-drifting data streams. In *Proceedings of the 10th ACM International Conference on Knowledge Discvoery and Data Mining (KDD)*. 128–137.

Freund, Y. and Schapire, R. E. 1996. Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning*. 148–156.

Gao, J., Fan, W., and Han, J. 2007. On appropriate assumptions to mine data streams: Analysis and practice. In *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM)*. 143–152.

Grizzard, J. B., Sharma, V., Nunnery, C., Kang, B. B., and Dagon, D. 2007. Peer-to-peer botnets: Overview and case study. In *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets (HotBots)*. 1–8.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. 2009. The WEKA data mining software: An update. *ACM SIGKDD Explor. 11,* 1, 10–18.

Hamlen, K. W., Mohan, V., Masud, M. M., Khan, L., and Thuraisingham., B. M. 2009. Exploiting an antivirus interface. *Comput. Stand. Interfaces 31,* 6, 1182–1189.

Hashemi, S., Yang, Y., Mirzamomen, Z., and Kangavari, M. R. 2009. Adapted one-versus-all decision trees for data stream classification. *IEEE Trans. Knowl. Data Engin. 21,* 5, 624–637.

Hulten, G., Spencer, L., and Domingos, P. 2001. Mining time-changing data streams. In *Proceedings of the 7th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 97–106.

Kolter, J. and Maloof, M. A. 2004. Learning to detect malicious executables in the wild. In *Proceedings of the 10th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 470–478.

Kolter, J. Z. and Maloof, M. A. 2005. Using additive expert ensembles to cope with concept drift. In *Proceedings of the 22nd International Conference on Machine Learning (ICML)*. 449–456.

Lemos, R. 2006. Bot software looks to improve peerage. *SecurityFocus*. www.securityfocus.com/news/11390.

Li, Z., Sanghi, M., Chen, Y., Kao, M.-Y., and Chavez, B. 2006. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 32–47.

Masud, M. M., Gao, J., Khan, L., Han, J., and Thuraisingham, B. 2008a. Mining concept-drifting data stream to detect peer to peer botnet traffic. Tech. rep. UTDCS-05-08, The University of Texas at Dallas, Richardson, Texas. www.utdallas.edu/ mmm058000/reports/UTDCS-05-08.pdf.

Masud, M. M., Gao, J., Khan, L., Han, J., and Thuraisingham, B. M. 2009. A multi-partition multi-chunk ensemble technique to classify concept-drifting data streams. In *Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD)*. 363–375.

MASUD, M. M., KHAN, L., AND THURAISINGHAM., B. 2008b. A scalable multi-level feature extraction technique to detect malicious executables. *Inf. Syst. Frontiers 10,* 1, 33–45.

MICHIE, D., SPIEGELHALTER, D. J., AND TAYLOR, C. C., EDS. 1994. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood Series in Artificial Intelligence. Morgan Kaufmann, 50–83.

NEWSOME, J., KARP, B., AND SONG, D. 2005. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 226–241.

QUINLAN, J. R. 2003. *C4.5: Programs for Machine Learning* 5th Ed. Morgan Kaufmann, San Francisco, CA.

RISH, I., GRABARNIK, G., CECCHI, G. A., PEREIRA, F., AND GORDON, G. J. 2008. Closed-Form supervised dimensionality reduction with generalized linear models. In *Proceedings of the 25th ACM International Conference on Machine Learning (ICML)*. 832–839.

SAJAMA AND ORLITSKY, A. 2005. Supervised dimensionality reduction using mixture models. In *Proceedings of the 22nd ACM International Conference on Machine Learning (ICML)*. 768–775.

SCHOLZ, M. AND KLINKENBERG, R. 2005. An ensemble classifier for drifting concepts. In *Proceedings of the 2nd International Workshop on Knowledge Discovery in Data Streams (IWKDDS)*. 53–64.

SCHULTZ, M. G., ESKIN, E., ZADOK, E., AND STOLFO, S. J. 2001. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 38–49.

STEWART, J. 2003. Sinit P2P trojan analysis. www.secureworks.com/research/threats/sinit.

TUMER, K. AND GHOSH, J. 1996. Error correlation and error reduction in ensemble classifiers. *Connect. Sci. 8,* 3, 385–404.

VX Heavens 2010. VX Heavens. vx.netlux.org.

WANG, H., FAN, W., YU, P. S., AND HAN, J. 2003. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the 9th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 226–235.

YANG, Y., WU, X., AND ZHU, X. 2005. Combining proactive and reactive predictions for data streams. In *Proceedings of the 11th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 710–715.

ZHANG, P., ZHU, X., AND GUO, L. 2009. Mining data streams with labeled and unlabeled training examples. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM)*. 627–636.

ZHAO, W., MA, H., AND HE, Q. 2009. Parallel *K*-means clustering based on MapReduce. In *Proceedings of the 1st International Conference on Cloud Computing (CloudCom)*. 674–679.

Received April 2011; revised July 2011; accepted August 2011