Scalable malware classification with multifaceted content features and threat intelligence

X. Hu
J. Jang
T. Wang
Z. Ashraf
M. Ph. Stoecklin
D. Kirat

Recent years have witnessed the very rapid increase in both the volume and sophistication of malware programs. Malware authors invest heavily in technologies and capabilities to streamline the process of building and mutating existing malware programs to evade traditional protection. One major challenge currently faced by the antivirus industry is to efficiently process the vast amount of incoming suspicious samples. Since most new malware is a variation of an existing malware family with the same forms of malicious behavior, automatic clustering and classification of malware programs into families have become valuable tools for malware analysts. Such grouping criteria not only allow analysts to prioritize the allocation of their investigation efforts but may also be applied to detect new malware samples based on their association with existing families. In this paper, we address the multi-class malware classification challenge from a scalability perspective. We present the design, development, and evaluation of a novel machine learning classifier trained on multifaceted content features (e.g., instruction sequences, strings, section information, and other malware features) as well as threat intelligence gathered from external sources (e.g., antivirus output). Our experiments on a dataset of 21,741 malware samples demonstrate the efficacy and precision of the proposed algorithm and also provide insights into the utility of various features.

Introduction

It is not news that malware is a huge and rapidly growing problem for enterprises, home users, and educational institutions, as well as health care providers, nation states, and government agencies. With the tools and services available to malware writers such as automatic malware generator and obfuscation services, traditional defense technologies can be easily defeated by rapid and on-the-fly generation of obfuscated, polymorphic, and metamorphic versions of malware [1]. One of the steps in dealing with the huge number of malware samples is to be able to classify binary executables into different malware groups, called families. This will help track various generations and advancement of a malware family as well as automatically detecting and classifying new binaries [2–6]. Machine learning has long been used in researching ways to automatically detect and classify binaries [7, 8]. Various approaches have been suggested; the success of the approach revolves around the choice of the features (i.e., inherent attributes of malware samples), speed of the feature extraction, algorithms that classify the binaries into families, and the quality of the training dataset.

In this paper, we present our approach to solving the Microsoft Malware Classification Challenge on the Kaggle Platform [9]. We were given two datasets: a training set and a test set with 10,868 and 10,873 malware samples, respectively. For each malware sample, two types

Digital Object Identifier: 10.1147/JRD.2016.2559378

of files are provided: hexadecimal byte content and Assembly Language Source (ASM) files produced by IDA Pro, a commercial disassembler software. We address the multi-class malware classification challenge from a scalability perspective. We present the design, development, and comparison between various machine learning classifiers trained on a combination of content features (e.g., instruction sequences, strings, section information) and threat intelligence gathered from external sources (e.g., antivirus output). We also propose several optimization methods to further improve the classification accuracy. Our experiments on the real-world malware datasets demonstrate the efficacy and precision of the proposed algorithm in processing large malware datasets and also provide insights into the utility of various features.

Related work

There are many previous works on malware classification and detection [3, 5, 8, 10, 11], including works on theoretical limits of malware detection [12, 13]. LeDoux and Lakhotia [14] describe machine learning techniques used for malware classification, where malware features are extracted from static and dynamic malware analysis. Most of the static approaches use *N*-gram-based feature extraction [4, 5, 15–18]. The *N*-grams are extracted from the machine bytecode, disassembled instructions, or instruction mnemonics. Jacob et al. [4] studied the preserved statistical similarity over packed binaries, and proposed a packer-agnostic bigram-based malware classification measure. *N*-grams can be suboptimal in finding similarity among permuted codes. To deal with this, Karim et al. [2] proposed a variation of *N*-gram called the "*n*-perm" method, which is similar to the *N*-gram method. However, the order of the characters is irrelevant. The *N*-gram approach usually produces high-dimensional features, making the problem intractable because of the amount of resources required. Jang et al. [5] proposed a feature hashing technique to reduce the high-dimensional feature space and evaluated it on *N*-gram-based features. Another approach to malware classification involves converting malware binaries into images and using signal processing techniques to extract features [19, 20]. Some previous works only used the attributes available in the Portable Executable (PE) file header [7, 21, 22]. When unpacked code is available with accurate disassembly, more robust features can be extracted from the semantics of the code [3, 5, 10, 23]. Code semantics-based features are robust because they can find similarity among polymorphic codes. Hu et al. [3] studied call-graphs-based features that are less susceptible to instruction-level obfuscations and proposed a scalable nearest-neighbor search technique for a large graph database of malware. Kruegel et al. [10] proposed a technique to extract the control-flow graph from network streams, convert it into canonical form, and perform similarity computations to detect polymorphic worms.

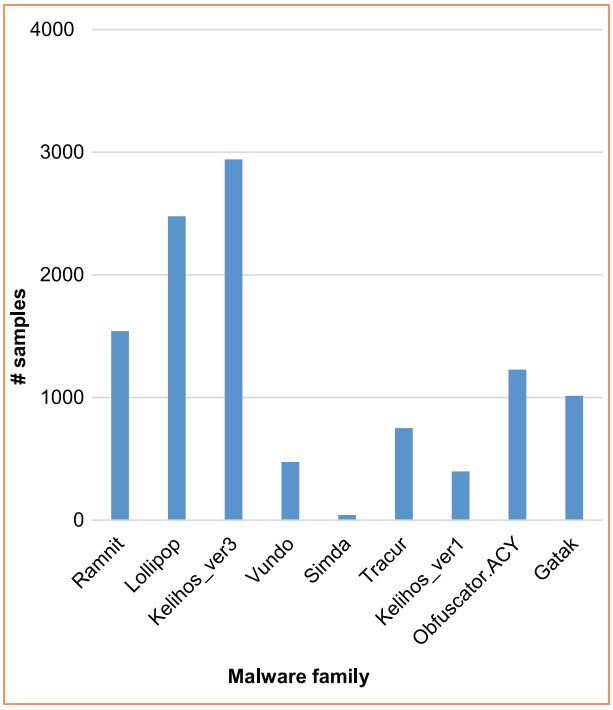

Figure 1

Distribution of malware families in the training dataset.

Several machine learning approaches such as association rule, support vector machine, decision tree, random forest, naive Bayes, and clustering techniques have been studied to build malware classifiers based on the features discuss above. Schultz et al. [7] first introduced data mining methods for detecting malware by using naive Bayes algorithms on strings and *N*-gram-based features. Gandotra et al. [8] has covered many of the interesting machine learning approaches. The work most closely related to ours in terms of machine learning approach is by Siddiqui et al. [11], which uses the random forest algorithm, along with bagging and decision tree classifiers.

Dataset overview

We investigate our methods with the malware dataset from the Microsoft Malware Classification Challenge on the Kaggle platform [9]. The dataset consists of nine different malware families: Ramnit, Lollipop, Kelihos ver3, Vundo, Simda, Tracur, Kelihos ver1, Obfuscator.ACY, and Gatak. Each malware sample provides raw binary content in a so-called hex dump of each sample excluding Portable Executable (PE) headers, and disassembled instructions and functions generated by the IDA Pro

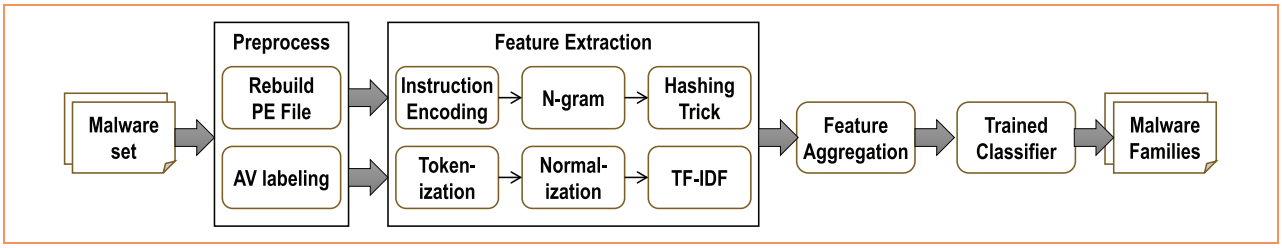

Figure 2

Overview of the system architecture.

disassembler. There are 10,868 malware samples for training and 10,873 malware samples for testing. **Figure 1** depicts the distribution of malware samples in each family. The distribution is non-uniform, with Kelihos having the most number of samples and Simda having the fewest. In addition, while inspecting the malware dataset, we notice that IDA outputs of some samples give little or no information. Manual investigation discloses that such samples are broken files or abnormal executable files, e.g., encrypted/compressed files, which creates additional challenges for our classification algorithm.

System architecture

Figure 2 summarizes the architecture of our system. At a high level, the malware analysis pipeline consists of four main components. First, it preprocesses the malware dataset to reconstruct the original PE files and obtains the labeling information from various AV (antivirus) software. Then, it extracts two types of features, i.e., machine code instruction features and AV label features, and performs necessary feature transformation. Next, these two types of features are aggregated into a single feature vector, which is used to train the classification model and finally determine the malware families in the test dataset.

Preprocessing

For PE files, we first reconstruct executable PE files, by converting hex dumps into raw binary, and by reconstructing PE headers based on the information from the IDA Pro outputs. This allows us to plug in a variety of analysis methods to comprehensively extract a rich set of features, including static and dynamic-analysis features and AV labels. Based on the PE file format [24], we extract necessary information from IDA outputs, such as virtual/raw sizes, addresses, flags of sections, and entry points. Since IDA outputs do not provide complete information to recover the original PE headers, we apply default values to some fields (e.g., `TimeDateStamp`), employ heuristics to extract information (e.g., the location of `public start`, `WinMain`, `DllMain` addresses for entry points), and arrange sections by handling "collapsed" parts where information was hidden in IDA Pro's outputs.

For non-executable samples, we reconstruct the original file by converting hex dumps into raw binary and by identifying the original file format and prepending truncated "headers" (available either in the hex dump or the IDA output). After reconstructing the original file, appropriate preprocessing for each file type is further required to obtain meaningful features. For example, (1) CryptFF (i.e., every byte XORed with 0xFF) files are decrypted by XOR-ing with 0xFF to recover the original file, (2) LHA/RAR/7zip compressed files are decompressed, (3) UPX-packed files are unpacked, (4) base64-encoded files are decoded, (5) malformed file signatures are fixed (e.g., the DOS header signature changed from "`MZ`", which are the initials of Mark Zbikowski, an MS-DOS developer, to "`mz`"—or the PE signature changed from "`PE`" to "`pe`"), and (6) executable file attachments are extracted from Microsoft Outlook** message files.
We then utilize IDA Pro to obtain disassembly instructions of the recovered/extracted executable files.

Feature extraction

AV label features

We use VirusTotal to obtain AV labels for reconstructed PE files. VirusTotal provides scanning results of over 40 different antivirus products. Antivirus scanners typically detect malware using signatures that would be built based on analysis of specific malware. Therefore, if two samples are detected by the same signature of the same AV scanner, both samples likely belong to the same malware family. However, there are some challenges with leveraging such intelligence. First, AV labels are not necessarily consistent across different AV scanners—even within the same AV scanner, e.g., different labeling conventions (`troj` vs. `Trojan`) and different variant names (`troj.aa` vs. `troj.1`). One way to address the issue is to tokenize AV labels based on punctuations (e.g., ., -, _), and normalize tokens to derive consistent malware family labels. Second, we may not be able to achieve

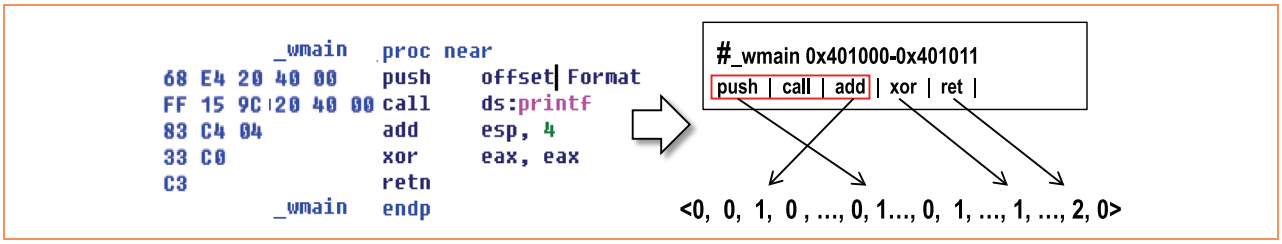

Figure 3

Encoding a function into a feature vector.

complete detection coverage. Although AV scanners are capable of handling some packed binaries, they may still miss some samples due to sophisticated obfuscation. In addition, since there could be errors in the process of reconstructing PE files, given limited available information from the IDA output, especially identifying entry points, AV scanners may fail to detect malicious code. Because of these limitations, the AV labels cannot be directly used to classify malware (at least unable to achieve the desired level of accuracy). Instead, in our experiments, we utilize AV label features as a supplement to other features, such as instruction features.

Instruction N-gram features

We use IDA Pro to disassemble the reconstructed malware binaries into a sequence of machine instructions, from which the features are extracted. One of the key components in the proposed algorithm is the similarity comparison between malware samples based on the disassembled instruction sequences, e.g., move eax, ebx; cmp eax, 1h. The main challenge in similarity comparison lies in handling the variations of machine instructions. Malware often undergoes changes for many reasons, such as mutation, polymorphism, and obfuscation where semantically equivalent instruction sequences are used to replace each other. As a consequence, ensuring exactness in comparing instructions does not tolerate any variation in the code syntax. At the other extreme, correctness is compromised if all forms of variation are tolerated. We strike a balance between these two extremes by using opcode sequences (move, cmp, etc.) as a succinct representation of the instruction semantics.

Using opcodes offers several unique benefits. Opcodes generalize well to represent variants within a malware family, because malware in the same family are often derived from the same code base, and thus share similarities in their machine instructions. However, due to relinking, rebinding, and rebasing, the operands (e.g., registers, memory addresses) of instructions tend to differ across variants. Using opcodes and ignoring the operands (i) makes the algorithm more resilient to low-level mutations while providing a meaningful characterization of malware semantics and (ii) reflects the functionality of malware programs in terms of the operations performed by their machine instructions. However, we also would like to point out that the N-gram opcode sequence analysis may still be circumvented by advanced obfuscating transformation such as VM (virtual machine)-based packing.

With this encoding scheme, a program is represented as a sequence of opcodes, e.g., move, push, pop, jmp, call, etc. Then, the challenge becomes how to convert the variable length instruction sequences into fixed length feature vectors suitable for model learning (see **Figure 3**). To better characterize the contents of malware programs, we use N-gram analysis, which moves a fixed-length window over an instruction sequence and considers a subsequence of length N at each position. The resulting N-gram of opcodes reflects short instruction patterns and implicitly captures the underlying program semantics. Next, for each malware program, a feature vector V is constructed in an $|S|$-dimensional space ($|S| = |\mathcal{O}|^N$ where $\mathcal{O}$ is the set of all possible opcodes). Each dimension of V is the number of *occurrences* of a particular opcode N-gram. In this way, the similarity between two malware programs (m, v) can be geometrically calculated as the Euclidean distance between their feature vectors in the vector space:

$$d(m,n) = \|V_m - V_n\| = \sqrt{\sum_{i=1}^{|S|}(V_m(i) - V_n(i))^2}.$$

Compared to the other similarity metrics (e.g., locality-based hashing), geometric calculation of similarity in the vector space provides *explicit feature representation* [25], where the importance or contribution of each N-gram in classifying malware can be traced back to its original code patterns. For N-grams that may correspond to inherent characteristics of a malware family (e.g., those that appear frequently within a family but rarely in others), their original code segments can be traced back and used as signatures to detect malware variants.

The N-gram analysis provides a useful feature representation for machine learning classifiers. However, one inherent problem of the resulting feature vectors is that they are very high dimensional, making storage and comparison of these feature vectors quite costly and computationally expensive. In this work, we address this "curse of dimensionality" with the *hashing kernel* technique. Kernel methods [26] are powerful tools used in machine learning to allow operation in the high-dimensional feature space without having to compute the coordinates of the data in that space. This is particularly useful when the input data has a non-linear decision boundary but can be linearly separated in a high dimensional feature space. In general, given input data $x_1, \ldots, x_n \in \mathcal{X}$ for some input domain $\mathcal{X}$, the kernel methods compare two input data as:

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$$

where ϕ is the mapping function from $\mathcal{X}$ to some feature space. In analyzing malware programs, however, we have encountered the opposite problem: the original space is very high-dimensional, implying that the input data are likely already linearly separable, and hence there is no need to map the input vectors to a higher-dimensional feature space. Since the number of dimensions D determines the complexity when computing the vector distance and D increases exponentially with N in the N-gram (i.e., $D = |\mathcal{O}|^N$, where $|\mathcal{O}|$ is the number of different opcodes and in practice $|\mathcal{O}| > 200$), even a small N like 3 will result in a (very sparse) feature vector with more than 8 million dimensions, which is computationally prohibitive when calculating similarities for a large amount of malware samples. Unfortunately, N must be at least 3 or 4 to be sufficiently descriptive for capturing the program semantics. To address these kinds of problems, the *hashing-kernel* has recently been developed in the machine learning community [27]. Instead of working on the original feature space, the technique *hashes* the high dimensional input vector $\mathbf{x} \in \mathbb{R}_n$ into a lower dimensional feature space $\mathbb{R}_m$ with the mapping function $\phi : X \rightarrow \mathbb{R}_m$. Since $m \ll n$, the hashing trick reduces a feature vector to a more compact representation, allowing the classification algorithm to handle a large volume of data and save both computation and memory requirements. Previous research has shown that the hash kernel approximately preserves the inner product between vectors and the penalty incurred from using a hash for reducing dimensionality only grows *logarithmically* with the number of samples and groups [27].

To use the hashing kernel, instead of assigning each N-gram a unique index, we apply a *uniform* hash function $H : \{N-\text{gram}\} \rightarrow [1 \ldots m]$ that hashes N-gram directly into a position in the feature vector of length m. In case of a collision where two or more N-grams map to the same position, the sum of the frequencies of all the colliding N-grams is used as the value in the new vector. More formally, for malware M and M^0, let v and v^0 represent their original feature vector extracted from the encoded opcode sequences and ξ denote the mapping from the N-gram $(o_1, o_2, \ldots, o_N) \in S$ to the index in v. We define the hash feature map ϕ as

$$\phi_i(i) = \sum_{l:H(o)=i, l \in S} v(\xi(o))$$

and the distance between M and M^0 as

$$d_\phi(M, M') = \|v - v'\|_\phi = \|\phi(v), \phi(v')\|.$$

The choice of m, the length of the low dimensional vector, is a trade-off between classification accuracy and storage overhead plus computational complexity. Choosing a smaller m results in a shorter vector length, and, thus, faster distance computation and smaller memory footprint to store malware features. However, decreasing m reduces the number of bins in which the hash function can place the different N-grams and consequently increases the collision possibility, leading to over-compression of features and negative impact of the classification accuracy. From our experience, a practical trade-off can be achieved with $m = 2^{10}$ to 2^{14}, and a larger m usually provides only marginal improvements of the classification accuracy while incurring exponentially increasing overhead in terms of storage and computation. In this work, we choose $m = 2^{13}$ as our default setting which results in an 8,192-dimensional feature vector.

String/PE header features

We also tested some simple features as well. In particular, we ran the strings utility on the hex and ASM files for all training set samples in a family. This gave us a set of strings per family. We then looked for a subset of strings from all the strings for a family that are unique to that family. We then computed how popular those strings are within the family in terms of the percentage of training samples that have those strings. Unfortunately, these numbers did not show any prominent strings, and we determined that they are not well suited for malware classification, likely due to obfuscation and the heavily polymorphic nature of malware samples.

Learning algorithms

In the previous sections, we have described techniques for extracting different features from malware programs. In this section, we will investigate several machine-learning-based classification algorithms with real-world malware samples to understand the

effectiveness of different algorithms in handling the diverse sets of malware families.

Performance metrics

Before discussing the concrete classification algorithms, we first introduce the metrics used in the Kaggle competition to evaluate the performance of various classification algorithms. This will help us gain insights into the underlying factors that drive the performance of different algorithms.

As described earlier, the dataset provided in the Kaggle competition consists of 9 distinct malware families. According to the rules, the submission (result file) should contain 10 columns: the first column is the sample ID (uniquely identifying each malware sample) and the remaining 9 columns contain the predicted probability (between 0 and 1) of the sample belonging to each of 9 malware families. Assuming there are N malware samples each belonging to *one* single malware family, the performance of a particular prediction result from a classification algorithm is evaluated using *multi-class logarithmic loss*:

$$\text{logloss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{9} y_{ij}\log(p_{ij})$$

where y_{ij} is 1 if the sample i is indeed in malware family j and 0 otherwise; and p_{ij} is the predicted probability of sample i belonging to family j. In order to avoid the extremes of the log function, predicted probabilities are replaced with

$$\max\left(\min(p, 1-10^{-15}), 10^{-15}\right).$$

Two observations can be made from the above definition. First, the range of logloss value is between 0 and 34.5, and the better the classification algorithm is, the closer its log loss value is to 0. However, due to the replacement of the extreme values, a logloss score of 0 is unachievable. Even if the classification algorithm makes the perfect prediction i.e., assigning probability 1 to the correct malware family, p_{ij} will be assigned the value $1–10^{-15}$, which incurs a non-zero logloss, albeit a very tiny one (i.e., -9.9920×10^{-16}). On the other hand, when the algorithm makes a completely wrong prediction, i.e., assigning probability 0 to the correct malware family, p_{ij} will be replaced with 10^{-15}, leading to a logloss value of $-\log(10^{-15}) = 34.5388$. Another important observation is that a completely wrong prediction would have a detrimental effect on the final logloss score. For instance, assigning probability 0 to the correct malware family will contribute a 34.53 logloss to the total metric. In contrast, a probability of 0.0001 will only create an addition of $-\log(0.0001) = 6.91$ logloss in case of the wrong prediction and a $-\log(0.9999) = 0.0001$ increase in case of the correct prediction. In other words, the probability assignment can be optimized as a function of the classification confidence, and unless absolutely certain, the assignment of probability 0 should be avoided. Having introduced the performance metric, in the rest of this section, we will describe several machine-learning-based classification algorithms that we have investigated in this study.

Weighted nearest neighbor classifier

One of the simpler classification algorithms is the k-nearest neighbor (KNN) classifier. KNN is a member of non-parametric methods that determines the classification results based on the closest examples in the feature space. The standard KNN classifiers output a *single* class membership, where an unknown sample is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. However, in the malware classification case, a multi-class classification algorithm is expected. For this, we developed a weighted multi-class classification adaptation of the standard KNN algorithm. More specifically, for each sample i in the test data, assume it has n matched training samples denoted as $r_1, r_2, \ldots, r_n$. Let $S_j(r_k)$ be the similarity score of sample k matched to malware family $j \in 1 \ldots 9$. Then p_{ij}, the probability of sample i belonging to family j, is computed as:

$$p_{ij} = \frac{\sum_{k=1}^{n} S_j(r_k)}{\sum_{k=1}^{n}\sum_{m=1}^{9} S_m(r_k)}$$

One of the main shortcomings of the KNN classifier is its sensitivity to the local structure of the data, and therefore, it does not perform well in the high-dimensional feature space where the sample distribution may be dispersed. In our experiments, the weighted KNN classifier achieves a 1.30 logloss value.

Logistic regression

We also tested a linear-model-based classification algorithm, i.e., logistic regression. Logistic regression is a special type of regression model that "tries" to predict a discrete variable ($y = 0$ or 1) with a set of known features x, by fitting a logistic function. More specifically, it learns a probability function of y given x in the form:

$$P(y = 1|x) = h_\theta(x) = \frac{1}{1 + \mathrm{e}^{-\theta^{\mathrm{T}}x}} \equiv \sigma(\theta^{\mathrm{T}}x)$$
$$P(y = 0|x) = 1 - h_\theta(x)$$

where $h_{\boldsymbol{\theta}}(z) \equiv 1/(1 + \mathrm{e}^{-z})$ is the sigmoid or logistic function, which maps the linear combination of feature value $\boldsymbol{\theta}^{\mathrm{T}}\mathbf{x}$ into the range of [0,1], which can be interpreted as the probability. Given a set of training examples,

logistic regression minimizes the following cost function to find the best coefficient $\boldsymbol{\theta}$:

$$J(\boldsymbol{\theta}) = -\Sigma\left(y^{(i)} \log\left(h_{\boldsymbol{\theta}}\left(x^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - h_{\boldsymbol{\theta}}\left(x^{(i)}\right)\right)\right)\right)$$

In this study, we evaluated the logistic regression algorithm using 5-fold cross validation on the entire training datasets. The training model took 2,190 seconds, and prediction took 0.06 seconds. Our experiment showed that the logistic regression model correctly predicted 98.2% of samples, resulting in a logloss value of 0.231.

Support vector machine

We further evaluated a non-linear learning algorithm called the support vector machine. The support vector machine (SVM) is a discriminative classifier model, which learns a hyperplane from the set of training data that best separates between positive and negative samples. SVMs address the problems of overfitting and capacity control associated with the classical learning machines and therefore tend to perform better in many scenarios. For a given learning task with a finite training set, an SVM strikes a balance between the accuracy obtained on the given training set and the generalization of the algorithm, which measures its ability to learn future unknown data without error. The flexible generalization ability of SVMs makes them suitable for real-world applications. More importantly, SVMs can efficiently learn non-linear separation boundaries by using a kernel function that implicitly maps the inputs sample data into high-dimensional feature spaces. We evaluated the SVM with Gaussian kernel (also known as RBF, or radial basis function, kernel) using the same 5-fold cross validation of training datasets. The training took 7,984 seconds, and the resulting classifier achieves 79.72% accuracy with a logloss value of 0.55812. The worse performance of the SVM comparing with simple linear logistic regression model could potentially be attributed to the sparsity of malware feature vectors.

Random Forest

Finally, we experimented on one type of ensemble learning methods called the Random Forest. The main idea of ensemble learning is to combine a set of "weak learners" to form a "strong learner." In case of random forests, the weak learners correspond to a set of decision trees constructed by using a random subset of the training data. At each candidate split in the learning process, the algorithm also selects a random subset of features to train the decision tree. More specifically, the tree growing process works as follows:

1. Given a set of N training samples, at each round, a subset of n samples is chosen at random with replacement from the training set. These samples will be the training set for growing the particular decision tree.
2. If there are M input variables (features), m variables are selected at random at each node, and the best split on these m variables is used to split the decision tree node.
3. Each tree is grown to the largest extent without pruning.

The above process repeats for every sub-tree in the forest until the number of trees reaches the predefined number k set by the user. Generally, a larger k leads to better accuracy in classification, but it also incurs higher computational cost. After training, predictions for new samples can be made by taking the votes from individual decision trees, and the forest chooses the class that has the most votes received from all trees in the forest. This ensemble approach often leads to better model performance, because it decreases the variance of the model (due to averaging) without increasing the bias of the forest. We tested the random forest classifier with 3,000 random trees and 4-gram features. The training took 2,362 seconds, and prediction took 35 seconds. Overall, random forests achieve a much higher accuracy at 98.89%, and the log loss value is 0.0774 without any tuning or optimization. Because of its promising performance, we choose random forests as our base classifier for further optimization.

Tuning/optimization

Feature selection/weighting

We use AV label features to further tune our classifier. As we discussed in the section "AV label features," different AV vendors may assign different AV labels for the same malware. Due to such inconsistencies across AV labels, we cannot simply group samples based on AV family names without normalization.

Given AV labels, we first remove any white spaces and make them to lowercases, and tokenize AV labels based on delimiters, such as `.`, `,`, `_`, `!`, `-`, `@`, `(`, `)`, `[`, `]`, `:`, and `/`. For example, the AV label `Trojan[Backdoor]/Win32.Shiz` yields the following tokens: `trojan`, `backdoor`, `win32`, and `shiz`. Then, we calculate tf-idf (term frequency-inverse document frequency) for every token to measure how important a token is to the 9 malware family names. tf-idf is often used to measure the significance of a term in a document by considering two statistics: (1) term frequency quantifies the weight of a term by calculating how many times a term appears in a document; and (2) inverse document frequency quantifies the rarity of a term by calculating the logarithm of the total number of documents divided by the number of documents where a term appears. For example, token `win32` appears very frequently and tf-idf of win32 in all 9 families are 0. On the other hand, tf-idf of token `tracurc` in the `Tracur` family is 0.47734, because the

token appears only in `Tracur` family. Note that although family name `Tracur` does not exactly match with token `tracurc` (i.e., extra "`c`" at the end), tf-idf captures its significance in the family. As a result, the family of a sample can be determined by choosing the family with the highest tf-idf sum of all tokens. In our experiments, we calculate tf-idf with both 1-gram and 2-gram tokens.

Combining multiple features

Because different types of features capture different aspects of malware programs, a single feature type is likely insufficient to characterize the entire spectrum of malware samples. For instance, instruction features are useful in detecting unknown malware programs that are variations of existing malware family. However, they may not be very effective in dealing with evasive techniques like obfuscation or runtime packing. On the other hand, AV label-based features are very well suited for existing malware samples that have been analyzed by various AV companies. However, the classification results are often unreliable if the malware program belongs to a new family or is simply a previously unseen variant. In such case, labels from different AV vendors are often inconsistent or contradictory. The lack of consensus created a major challenge for classification algorithms. Exploiting this complementary nature among different features, we aggregate multiple features into a composite feature vector. There are many possible ways to combine multiple features, and in this study we used a simple approach that concatenates the original vectors of features into a single feature vector. The increasing dimensionality makes feature selection an essential step in order to avoid overfitting and improve model performance. Fortunately, random forests directly perform feature selection during the process of building sub-trees and classification rules and therefore are well suited for the task. In total, the combined feature vector consists of 13,547 features, and training random forest took 2,867 seconds. In addition, we also performed tuning and optimization on the classification algorithm, and the results will be presented in the next section.

Evaluation

In this section, we evaluate the performance of our malware classification algorithm. We first look at the effectiveness of reconstructing PE files for malware programs. This is a critical prerequisite for obtaining reliable AV label features. Then, we will describe the experiment results of applying the classification algorithm on real world malware samples in the test dataset.

Effectiveness of PE reconstruction

While reconstructing headers and files from the hex dump and IDA outputs, we noticed that some files were non-executable files and required appropriate processing (see the section "Preprocessing"). A detailed breakdown of file types identified by our investigation is described in **Table 1**. There were several challenges in the process of file reconstruction. For example, the IDA output of some files (e.g., collapsed) lacked critical information, such as section and entry point. In addition, some binary files were "broken," and no useful information about sections and files was available. Some Nullsoft Scriptable Install System files and compressed files were truncated such that the actual binary content could not be extracted. "Empty" files had only "`??`" characters in their binary content, and no information about samples was available. Instead, MD5 checksum of the original file was given, and we use the MD5 to obtain AV labels from VirusTotal.

Table 1 File types. (EXE: executable; DLL: dynamic link library; LHA: LHarc; RAR: Roshal archive.)

Types	*# training samples*	*# testing samples*
EXE, DLL	10,806	10,806
CryptFF	21	23
Malformed file signature	5	5
LHA compressed	9	9
RAR compressed	13	12
7zip compressed	-	2
Outlook message	1	-
Octet stream	-	1
Based64 encoded	1	1
Scrambled binary	1	1
Binary without section info	3	-
"Empty" file	8	13
Total	**10,868**	**10,873**

Performance evaluation

One of the core steps in training machine learning models is to select appropriate values for hyperparameters. In contrast to regular parameters in the learning algorithm, which are optimized and selected automatically during the training process, hyperparameters are those that need to be specified by the user. The proper choice of these hyperparameters is often critical to ensure that the model does not overfit or underfit the training data. In this study, we optimized the selection of several hyperparameters in the random forest using grid search, which exhaustively searches through a subset of the hyperparameter space to determine the best combination of parameter values. The search is guided by the performance metric computed using the k-fold cross validation on the training datasets.

Table 2 Evaluation results on different features.

	AV label features	*Instruction features*	*Combined*
5-fold CV accuracy on training data	0.994	0.993	0.998
Logloss on test data	0.0695553	0.0546879	0.0258737
Training time (seconds)	1533	728	2867

Table 3 Probability assignments of malware samples to malware families.

Samples	*Malware family probability (every column represents one family)*								
Sample 1	0.984	0.006	0	0.001	0	0.001	0.002	0.004	0.002
Sample 2	0	1	0	0	0	0	0	0	0
Sample 3	0	0	0	0	0	0	0	1	0
Sample 4	0	0.001	0.001	0.006	0.002	0.973	0	0.013	0.004
Sample 5	0.001	0	0.001	0.994	0	0.001	0.001	0.002	0
Sample 6	0	0	0	0.002	0.001	0.002	0	0.991	0.004
Sample 7	0.001	0	0	0.003	0	0.002	0	0.007	0.987

Specifically, grid search was used to select the following three main parameters of the random forest classifiers:

- *Number of estimators*—number of trees in the forest. We search the value range between 500 and 10,000. In general, a higher number of estimators leads to better accuracy. However, it also poses a time/quality trade-off.
- *Maximum number of features*—the number of features to consider when looking for the best split in the tree building step. Assuming the total number of features in the feature vector is n, we tested several choices between 1,000 to maximum n features.
- *Split criterion*—different methods for measuring the quality of a split. Two methods were incorporated into the search are Gini impurity and entropy.

Overall, our experiments showed that the parameter combination (1,000 estimators, 2,000 features, and "Gini" split criteria) appeared to strike a good balance between classification accuracy and the computational overhead, allowing us to quickly compare the effectiveness of various features and parameter settings.

We evaluated the classifier performance using both individual types of features and the combined feature. **Table 2** summarizes the comparative results. Both AV label and instruction based features perform fairly well on the training dataset in terms of the 5-fold cross validation (CV). This indicates that strong correlation exists within the same malware families, thus allowing the learning algorithm to select distinguishing features unique to a particular malware family. Nevertheless, there are still difficult samples that are misclassified, leading to the logloss of 0.069 and 0.054 respectively. Combining the two feature types offers a more comprehensive picture of malware characteristics. Therefore, not only does it improve the accuracy of 5 fold cross-validation on the training data, but it also dramatically reduces the logloss by more than 50% (i.e. from more than 0.054 to 0.0258) for the testing data. This demonstrates that the combined feature vector generalizes well to the previously unseen samples.

While investigating the classification results, we have noticed one potential optimization to improve the logloss result that is to adjust the probability assignment to various families. We found that a majority of the correctly classified samples have their probability distributed across multiple families. In **Table 3**, we display typical probability assignments of several samples.

We can see that the family with probability close to 1 is almost certain to be the correct family for the malware sample. In such cases, diverting some probability mass, even a small one, to other families is not an optimal strategy, because this will negatively affect the logloss value. Based on this observation, we devised a new probability assignment strategy, which would concentrate all the probability mass to a single family for those confident classification results. However, the main challenge is how to determine a good threshold such that if the probability of a sample belonging to a particular family is higher than the threshold, we will assign probability 1 to that family and 0 to others. As discussed earlier, according to the formula of logloss computation, the penalty of assigning probability 1 to the wrong malware family is quite significant. Therefore,

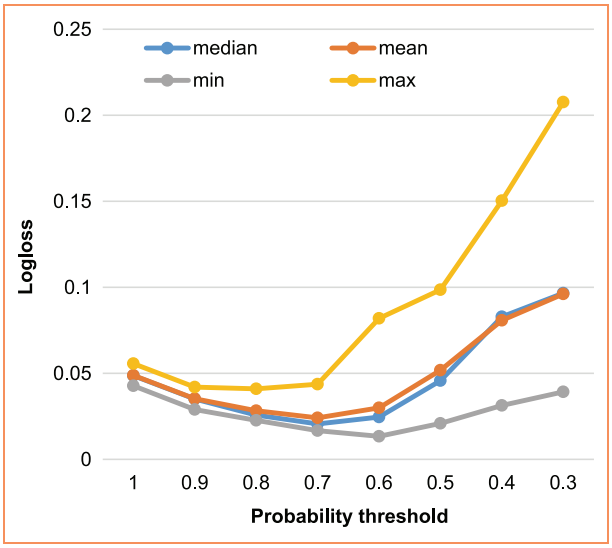

Figure 4

Optimizing for the probability threshold.

it is of critical importance that we choose the proper threshold value.

We use the labeled training dataset to guide the threshold selection process. Essentially, we vary the threshold between 0.1 and 1, and calculate the logloss value for each threshold using the training data. The correlation between probability threshold and logloss value is plotted in **Figure 4**. The experiments were repeated multiple times with different splits of training datasets and the median, mean, min, and max value of logloss are plotted in the figure. We can observe that the curve reaches the bottom around 0.6 and 0.7. As a result, we set the threshold to be 0.65 and apply it to the classification results of the test dataset. This, in fact, helped us reduce logloss by another 50% and achieve a value of 0.0121859.

Conclusion

In this paper, we have presented the design, implementation, and evaluation of a malware classification system based on multifaceted features such as machine instruction and AV label features. By exploiting the complementary nature among these features, the proposed system can accurately and efficiently classify unknown malware samples into specific families. The system extracts an aggregated feature vector from each malware program based on opcode representation and the intelligence from antivirus software. It then trains and optimizes a random forest classifier to learn the unique set of features that best distinguishes between malware families. To ensure the scalability and accuracy, we adopted a combination of a hashing kernel that reduces the dimensionality of feature vectors and an optimal probability assignment strategy. Equipped with these techniques, the system is experimentally shown to be able to accurately classify more than 10,000 malware samples with 99.8% accuracy in fivefold cross-validation and a logloss value of 0.0258.

References

1. "Annual threat report," FireEye Inc., Milpitas, CA, USA, 2015. [Online]. Available: https://www2.fireeye.com/WEB-2015RPTM-Trends.html
2. M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *J. Comput. Virol.*, vol. 1, no. 1, pp. 13–23, 2005.
3. X. Hu, T. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. ACM CCS*, 2009, pp. 611–620.
4. G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna, "A static, packer-agnostic filter to detect similar malware samples," in *Proc. DIMVA*, 2013, pp. 102–122.
5. J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Feature hashing malware for scalable triage and semantic analysis," in *Proc. ACM CCS*, 2011, pp. 309–320.
6. A. Mohaisen, O. Alrawi, M. Larson, and D. McPherson, "Towards a methodical evaluation of antivirus scans and labels," in *Information Security Applications*. Berlin, Germany: Springer-Verlag, 2014, pp. 231–241.
7. M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proc. IEEE Symp. Security Privacy*, 2001, pp. 38–49.
8. E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *J. Inf. Security*, vol. 5, pp. 56–64, 2014.
9. "Microsoft malware classification challenge," Kaggle, San Francisco, CA, USA. [Online]. Available: https://www.kaggle.com/c/malware-classification
10. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proc. Symp. RAID*, 2005, pp. 207–226.
11. M. Siddiqui, M. C. Wang, and J. Lee, "Detecting internet worms using data mining techniques," *J. Syst., Cybern. Informat.*, vol. 6, no. 6, pp. 48–53, 2008.
12. D. M. Chess and S. R. White, "An undetectable computer virus," in *Proc. Virus Bull. Conf.*, 2000, vol. 5, pp. 1–8.
13. F. Cohen, "Computer viruses: Theory and experiments," *Comput. Security*, vol. 6, no. 1, pp. 22–35, 1987.
14. C. LeDoux and A. Lakhotia, "Malware and machine learning," in *Intelligent Methods for Cyber Warfare*. Berlin, Germany: Springer-Verlag, 2015, pp. 1–42.
15. T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Proc. IEEE COMPSAC*, 2004, pp. 41–42.
16. J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *J. Mach. Learn. Res.*, vol. 7, pp. 2721–2744, Dec. 2006.
17. I. Santos, Y. Penya, J. Devesa, and P. Bringas, "N-grams-based file signatures for malware detection," in *Proc. ICEIS*, 2009, pp. 1–4.
18. R. Perdisci and A. Lanzi, "McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *Proc. ACSAC*, 2008, pp. 301–310.
19. L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proc. ACM VizSec*, 2011, pp. 1–7.

20. D. Kirat, L. Nataraj, G. Vigna, and B. S. Manjunath, "SigMal: A static signal processing based malware triage," in *Proc. ACSAC*, Dec. 2013, pp. 89–98.
21. M. Shafiq, S. Tabish, F. Mirza, and M. Farooq, "PE-Miner: Mining structural information to detect malicious executables in realtime," in *Proc. Symp. RAID*, 2009, pp. 121–141.
22. G. Wicherski, "peHash: A novel approach to fast malware clustering," in *Proc. USENIX Workshop LEET*, 2009, pp. 1–8.
23. E. Carrera and G. Erdélyi, "Digital genome mapping-advanced binary malware analysis," in *Proc. Virus Bull. Conf.*, 2004, pp. 187–197.
24. M. Pietrek, *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms809762.aspx
25. K. Rieck, *Malheur-Automatic Analysis Of Malware Behavior*, 2010. [Online]. Available: http://www.mlsec.org/malheur
26. J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
27. Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, A. Strehl, and V. Vishwanathan, "Hash kernels," in *Proc. 12th Int. Conf. Artif. Intell. Statist.*, 2009, pp. 496–503.

Received October 13, 2015; accepted for publication November 9, 2015

Xin Hu *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (huxin@us.ibm.com)*. Dr. Hu is a Research Scientist on the Security Services team at the IBM T. J. Watson Research Center. He received his Ph.D. degree in computer science and engineering from University of Michigan, Ann Arbor, in 2013. His research focuses primarily on cybersecurity, with an emphasis on big data security analytics and network security.

Jiyong Jang *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (jjang@us.ibm.com)*. Dr. Jang is a Research Scientist on the Security Services team at the IBM T. J. Watson Research Center. He received his Ph.D. degree in electrical and computer engineering from Carnegie Mellon University in 2013. His research interests include most areas of computer security, with an emphasis on software and network security. His current research focuses on big data security analytics and its applications to malware analysis, network security, and Web security.

Ting Wang *Lehigh University, Bethlehem, PA 18015 USA (ting@cse.lehigh.edu)*. Dr. Wang is currently an Assistant Professor in the Computer Science and Engineering department at Lehigh University. Before joining Lehigh, he was a Research Staff Member at IBM T. J. Watson Research Center. He received his Ph.D. degree from Georgia Institute of Technology in 2011. His research interests include data mining, machine learning, and privacy and security. His current research focuses on designing privacy-preserving and secure systems, quantitative analysis of privacy and security in emerging computing systems, and theoretical foundations of computational privacy.

Zubair Ashraf *IBM X-Force Advanced Security Team, Markham, ON L6G 1C7, Canada (zashraf@ca.ibm.com)*. Mr. Ashraf is security researcher and team lead for IBM X-Force Advanced Research. He is passionate about fighting all malicious activities in cyberspace [i.e., cyber-crime/attacks, advanced persistent threats (APTs), etc.]. Currently, he contributes to this via several means, such as actively and passionately educating and training others via his Twitter** account; blogging or presenting at security events; analyzing exploitation techniques, malware and vulnerabilities; and advising the IBM Security System product development teams on prevention and detection strategies.

Marc Ph. Stoecklin *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (mpstoeck@us.ibm.com)*. Dr. Stoecklin is the Manager of the Security Services Group, where he works on cyber security analytics with a particular focus on network, device, and industrial control system security, as well as big data analytics and visualization. He is an expert in behavior modeling and anomaly analytics based on machine learning and data mining techniques. Dr. Stoecklin holds an M.Sc. degree (2007) and Ph.D. degree (2011) in computer, communication, and information sciences from École Polytechnique Fédérale de Lausanne. Dr. Stoecklin is a member of the Institute of Electrical and Electronics Engineers.

Dhilung Kirat *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (dkirat@us.ibm.com)*. Dr. Kirat is a Research Scientist on the Security Services team at the IBM T. J. Watson Research Center. He received his Ph.D. degree in computer science from University of California, Santa Barbara, in 2015. His research interests are in areas of computer security, in particular malware analysis and security analytics.