Be Sensitive to Your Errors: Chaining Neyman-Pearson Criteria for Automated Malware Classification

Guanhua Yan
Department of Computer Science
Binghamton University, State University of New York
Binghamton, NY, U.S.A.
ghyan@binghamton.edu

ABSTRACT

Thwarting the severe threat posed by the voluminous malware variants demands effective, yet efficient, techniques for malware classification. Although machine learning offers a promising approach to automating malware classification, existing methods are oblivious of the costs associated with the different types of errors in malware classification, i.e., false positive errors and false negative errors. Such treatment adversely affects later applications of per-family malware analysis such as trend analysis. Against this backdrop, we propose a unified cost-sensitive framework for automated malware classification. This framework enforces the Neyman-Pearson criterion, which aims to maximize the detection rate under the constraint that the false positive rate should be no greater than a certain threshold. We develop a novel scheme to chain multiple Neyman-Pearson criteria on heterogeneous malware features, some of which may have missing values. Using a large malware dataset with labeled samples belonging to 12 families, we show that our method offers great flexibility in controlling different types of errors involved in malware classification and thus provides a valuable tool for malware defense.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and protection

Keywords

Malware, classification, Neyman-Pearson criterion

1. INTRODUCTION

The Internet is now inundated with numerous malware which are responsible for a wide range of malicious activities such as email spamming, botnets, and identity theft. As evidenced by the Symantec reports showing as many as 286 million malware variants were created in 2010 and 400 million in 2011 [35], we must equip ourselves with effective, yet scalable, solutions to mitigating the ever-growing malware threats. Traditional approaches based on malware signatures produced from AV (Anti-Virus) software companies have been shown fruitless in defending against current malware threats; some reports even go as far as claiming that the existing AV solutions are "dead" [11, 2].

In some previous efforts, machine learning has been offered as an alternative approach to classifying the vast volume of malware attacks [30, 14, 25, 27, 21, 32, 26, 1, 38]). Albeit promising, a few technical obstacles still remain as we apply machine learning to automated malware classification in practice. First, the numbers of malware variants belonging to different malware families are highly skewed [28]. Severe class imbalance across malware families poses significant challenges to controlling the accuracy of malware classification, as for a rare family with only a few instances, simply flagging every instance as negative would achieve high classification accuracy. Second, heterogeneous malware features can be extracted from malware programs [40]. Ideally, we would want to combine various types of features to achieve highly accurate malware classification. Existence of missing feature values hinders the deployment of standard machine learning techniques in a straightforward manner. Last but not least, malware samples that are accurately labeled are not easy to obtain, as it is difficult, if not impossible, to manually label a large number of malware variants, and it is hard to overcome the inconsistency among classification results by multiple AV software to find unbiased labeled samples by consensus [16, 18]. Hence, in many cases, we possess only a small number of malware samples that are confidently labeled along with a large number of unlabeled ones.

In this work we propose a unified malware classification framework that overcomes these challenges. This framework is built on an ensemble of cost-sensitive malware classifiers, each of which is trained individually on a type of features extracted from malware programs. Hence, even if we cannot collect one specific type of features from a malware program, the framework can still rely on other types of features to infer the family it should belong to. The cost-sensitive nature of individual classifiers offers us the flexibility of imposing different penalties on false positive and false negative errors. Taking advantage of how much penalties we assign to each type of classification errors, we enforce the Neyman-Pearson criterion [31, 4], which aims to maximize the detection rate of the ensemble classifier while ensuring that the false positive rate should be no greater than a certain threshold. When only a small number of malware samples are labeled, we leverage the structural information inherent

ASIA CCS'15, April 14–17, 2015, Singapore.

http://dx.doi.org/10.1145/2714576.2714578.

in a large amount of unlabeled data to train high-quality classifiers based on semi-supervised learning.

In a nutshell, our contributions made in this work can be summarized as follows. **(1)** We design and implement a unified cost-sensitive framework for automated malware classification, which overcomes various challenges we face in classifying large numbers of malware variants into their corresponding families in practice, such as missing feature values, class imbalance, and difficulty in obtaining accurately labeled samples. **(2)** To combine classification results from multiple cost-sensitive classifiers, we extend a well-established concept in hypothesis testing, Neyman-Pearson criterion [31, 4], and propose the *chain Neyman-Pearson criterion*, which can be used to ensure that the false positive rate of the ensemble classifier should be no greater than a certain threshold while maximizing its detection rate. **(3)** We apply the dynamic programming technique to search optimal configurations that satisfy the chain Neyman-Pearson criterion for individual classifiers, each trained on a specific type of malware features. **(4)** Using a malware dataset containing tens of thousands of malware instances belonging to 12 families, we demonstrate that our method offers great flexibility in controlling different types of errors involved in automated malware classification.

The rest of the paper is structured as follows. Section 2 presents the challenges in malware classification. Section 3 introduces a unified malware classification framework. Section 4 discusses how to train individual classifiers and Section 5 how to build an ensemble classifier. Section 6 shows the evaluation results. Section 7 discusses related work.

2. REALITY CHECK

Our study is based on a malware dataset from Offensive Computing [22] with 526,179 samples. While processing this dataset, we encounter the following three major challenges.

2.1 Labeled and unlabeled data

In order to know the family each malware variant belongs to, we upload its MD5 to the VirusTotal website [37] and obtain the detection results from 43 AV software. Among all these results, we consider only the detection results from the five major AV software, `McAfee`, `Kaspersky`, `Microsoft`, `ESET (NOD32)`, and `Symantec`. Next, we extract the malware family information from the detection result from each AV software. For instance, if Microsoft detects a malware program as `Trojan:Win32/Vundo.BY`, we then identify its family name as `Vundo`. Thereafter, we use the majority rule to label a malware instance: if four out of five AV software classify it as the same malware family, we assume it belong to that family. Using this method, we are able to label only 26,848 instances, which belong to 12 distinct malware families, `Bagle`, `Bifrose`, `Hupigon`, `Koobface`, `Ldpinch`, `Lmir`, `Rbot`, `Sdbot`, `Swizzor`, `Vundo`, `Zbot`, and `Zlob`. Conventional wisdom is that we train a classifier for each family based on only labeled data. The large amount of unlabeled data however contain rich structural information that can be further exploited to improve classification accuracy.

2.2 Class imbalance issue

Figure 1 shows the number of instances labeled in each family. The `Full` case includes both packed and unpacked instances, and the `Unpacked` case has only unpacked instances. The plot clearly shows high imbalance among different malware families. For instance, for the `Unpacked` case, the `Hupigon` family has 31.2 times as many instances as the `Bagle` family has. The class imbalance issue complicates the search for an optimal classifier [10]. For instance, consider a widely used measure, *classification accuracy*, which is defined to be the fraction of correctly classified instances. When we train an optimal classifier that maximizes classification accuracy from a dataset with only a few positive samples, a dummy classifier that simply classifies every instance as negative may stand out as the best one.

2.3 Missing feature values

We extract the following types of features from each malware program in our dataset. **(1) Hexdump 2-gram:** We use utility `hexdump` to produce byte sequences from each malware program, and a hexdump 2-gram feature is constructed by obtaining the frequencies of any two consecutive bytes in the program. **(2) Objdump 1-gram:** We use utility `objdump` to disassemble each malware program, and treat the concatenation of the prefix and the opcode in each instruction as a feature. The value of a feature is the frequency which which it appears in the program. **(3) PE header:** We extract information from the PE header of each malware program with utility `pefile`. We extract two types of features from PE headers. `PE-num`: numerical features extracted from PE headers, `PE-bool`: boolean features extracted from PE headers, such as bits in characteristic fields, whether a DLL is imported, and whether a system call in a DLL file is imported. **(4) PIN trace:** We execute each malware program in a controlled environment for five minutes and use Pin [12], a dynamic binary instrumentation tool, to dump the execution traces. We also extract two types of features from PIN traces. `PIN 2-gram`: the frequency of the ordered combination of opcodes in every two consecutive instructions, and `PIN SysCall`: the number of times that a system call has been invoked.

Figure 2 depicts the fraction of unpacked malware samples in each family that we are able to extract feature values successfully. Not surprisingly, we are able to extract `hexdump 2-gram` and `PE header` (including both `PE-num` and `PE-bool`) features successfully from every malware program. However, for those `objdump 1-gram` features, we cannot extract features from a significant portion of malware instances because objdump crashes during the disassembly process. This occurs similarly to Pin when we try to extract `PIN trace` (including both `PIN 2-gram` and `PIN SysCall`) features. Interestingly, there is no strong correlation between missing `objdump 1-gram` and `PIN trace` feature values: for some malware families (*e.g.*, Koobface, Zbot and Zlob), we are able to extract `objdump 1-gram` features from the majority of malware instances but can only extract `PIN trace` features from a small portion of malware instances.

3. A UNIFIED MALWARE CLASSIFICATION FRAMEWORK

A unified malware classification framework is illustrated in Figure 3. It works on a malware database that contains not only labeled malware programs but also a large number of unlabeled malware instances. Labeled malware samples can come from those identified manually by malware forensic analysts, or from consensus among multiple AV software. Due to the voluminous malware variants, many malware instances will remain as unlabeled in the malware database. Rather than ignoring these unlabeled samples, our frame-

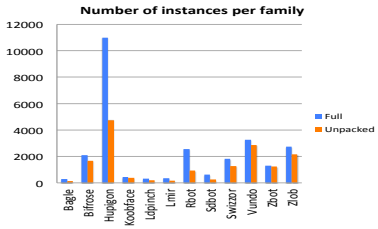


Figure 1: Imbalanced number of instances per family

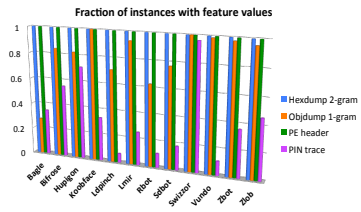


Figure 2: Fraction of instances with feature values per family

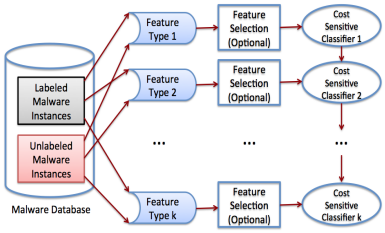


Figure 3: A unified malware classification framework

work exploits the structural information inherent among these unlabeled samples when training malware classifiers.

We extract various types of features from malware programs. There have been a large literature dedicated to discovering powerful features for malware detection or classification. In principle, our framework can integrate all these different types of features together, and provide a high-quality malware classifier based on their collective efforts. For each type of features extracted from malware programs, we may need to perform feature selection before training a classifier on them, because for some types of malware features, the number of features is so large that a classifier trained on all of them does not perform efficiently in practice, and for some classifiers, having more features does not necessarily mean that its performance is better than that when only a small number of features are used [40].

Once we have decided what features to use for each feature type, we use these features collected from both labeled and unlabeled malware instances to train a malware classifier. In parlance of machine learning, this process is semi-supervised learning. Semi-supervised learning contrasts with supervised learning, which relies on only labeled data when training a classifier. When labeled data are costly to obtain, the performance of supervised learning usually suffers because the distribution of labeled data used in the training dataset may not be representative of the true distribution of the new instances coming later. By contrast, semi-supervised learning exploits the structural information inherent in the large amount of unlabeled data to approximate better the true distribution of the instances which we will need to classify later. A fundamental assumption behind semi-supervised learning is that if two instances appear in the same cluster, they are likely to belong to the same class [6]. Based on this assumption, a semi-supervised classifier either propagates labeling information from labeled instances to those unlabeled ones belonging to the same cluster, or searches for classification boundaries through only sparse areas.

Another key component of our framework is the cost sensitiveness of the malware classifier we train on each type of features. When searching for an optimal classifier for a specific type of malware features, we apply the Neyman-Pearson criterion, which tries to maximize the detection rate under the constraint that the false positive rate must be no greater than a certain threshold [31, 4]. To enforce the Neyman-Pearson criterion, we use a cost-sensitive classifier with adjustable penalty weights on different types of classification errors. We search for an optimal setting from the parameter space of these weights, and use cross-validation techniques to ensure that the Neyman-Pearson criterion should be met.

With multiple individual classifiers, each of them may have its own opinion when we apply it on a new malware variant. As we may have missing features for a feature type, we assume that its corresponding classifier classifies it as negative. This naturally leads to a classifier ensemble based on the 'OR' rule: if any classifier decides that a new variant should belong to a specific malware family, the ensemble of classifiers believes it is a variant of that family; only all classifiers decide it is a negative sample does the ensemble classifier classifies it as negative. Such an 'OR' decision rule can be easily translated into a *sequence* of malware classifiers, each of which is trained on a specific type of malware features. If any of the classifiers decides a new sample is positive, the ensemble classifier terminates by flagging it as positive; only if the malware variant passes the tests of all individual classifiers can it be flagged as negative.

We further propose the chain Neyman-Pearson criterion, which is applicable to multiple classifiers that work in tandem. Under the chain Neyman-Pearson criterion, a dynamic programming method is used to spread the overall false positive rate allowed for the ensemble classifier over all the individual cost-sensitive classifiers as the constraints on their false positive rates.

We now theoretically analyze the ensemble classifier based on the 'OR' rule. Regarding the 'OR' rule that combines multiple classifiers, we have the following proposition about its Vapnik-Chervonenkis (VC) dimension, which measures the capacity of a classifier [36]:

Proposition 1. *Consider a hypothesis space H over domain $\mathbf{X} = \{0,1\}^n$, which is a set of $\{0,1\}$-valued functions. For each $h \in H$ and $\mathbf{x} = (x_1, x_2, ..., x_n) \in \mathbf{X}$, we have $h(\mathbf{x}) = \prod_{i=1}^{n} x_i$. The VC-dimension of H is then 2.*

Proof. Consider two points in $\mathbf{X}$, $\mathbf{x}_0 = \mathbf{0}$ and $\mathbf{x}_1 \neq \mathbf{0}$. Clearly, any $h \in H$ can shatter the two points. However, for any three distinct points in $\mathbf{X}$, there must be at least two of them each containing at least one non-zero element. No hypothesis in H can shatter these two points.

We further have the following about the VC-dimension of the classifier ensemble combined with the 'OR' rule.

Proposition 2. *Consider a set of k binary classifiers, whose VC-dimensions are summed up to d. The VC-dimension of the classifier ensemble combined with the 'OR' rule is upper bounded by $(d+2)\log_2[3e(k+1)^2]$.*

Proof. The k binary classifiers combined with the 'OR' rule can be deemed as a feed-forward architecture with $k+1$ computation nodes. The sum of all the computation nodes in this architecture is $d+2$ (note that the node corresponding to the 'OR' rule has a VC of 2, as seen from Proposition 1). According to Theorem 1 in [3], we have the number of realizable functions with m points, denoted by $\Delta(m)$, as follows:

$$\Delta(m) \leq ((k+1)em/(d+2))^{d+2}, \text{for } m \geq d+2. \quad (1)$$

We next show that if $m = (d+2)\log_2[3e(k+1)^2]$, it holds that $\Delta(m) < 2^m$. Note that:

$$\Delta((d+2)\log_2[3e(k+1)^2]) = [(k+1)e\log_2[3e(k+1)^2]]^{d+2}.$$

On the other hand, we have:

$$2^{(d+2)\log_2[3e(k+1)^2]} = [3e(k+1)^2]^{d+2}. \quad (2)$$

As we have both $\log_2(3e) < 3$ and $\log_2(k+1)^2 < 3k$ for $k \geq 1$. We thus have $\Delta(m) < 2^m$ when $m = (d+2)\log_2[3e(k+1)^2]$. Hence, the VC-dimension of the ensemble classifier is upper bounded by $(d+2)\log_2[3e(k+1)^2]$.

Therefore, the VC-dimension of the ensemble classifier is dominated by factor $d+2$, suggesting that the simple 'OR' rule does not improve significantly the capacity of the original set of classifiers, which agrees well with our intuition. Hence, our malware classification framework relies on individual classifiers that are sufficiently complicated in dividing malware families. To this end, we use the popular SVM classifier, whose VC dimension can be very high or even infinite when equipped with the radial basis kernel [5].

4. TRAINING INDIVIDUAL CLASSIFIERS

With the set of features chosen for each type of malware features, we prefer a classifier that is capable of not only performing semi-supervised learning but also having tunable parameters to impose different costs associated with different types of classification errors. Due to the extensibility of SVM for achieving both semi-supervised and cost-sensitive learning, it is used as the bedrock of individual malware classifiers in our framework. Consider two datasets, $\mathcal{D}_l = \{\langle \mathbf{x_i}, y_i\rangle\}_{i=1,...,n_l}$ where $y_i \in \{+1,-1\}$ and $\mathcal{D}_u = \{\mathbf{x_j^*}\}_{j=1,...,n_u}$, which contain labeled and unlabeled data, respectively. In order to train an SVM-type of classifier, we first project $\mathbf{x_i}$ or $\mathbf{x_j^*}$ into a Hilbert space with a mapping $\Phi : \mathbb{R}^d \to \mathcal{H}$, where d is the dimension of the feature space. In this Hilbert space, we search for a hyperplane with parameters $\mathbf{w}$ and b that minimizes the sum of margin errors, which are represented as ξ_i and ξ_j^*, respectively, for labeled and unlabeled training samples. Assuming that $I_+ = \{i : y_i = +1\}$, $I_- = \{i : y_i = -1\}$, $J_+ = \{j : y_j^* = +1\}$ and $J_- = \{j : y_j^* = -1\}$, the objective function is given by:

$$\begin{aligned}\min_{\mathbf{w},b,\xi,\xi^*,\mathbf{y}^*} V(\mathbf{w},b,\xi,\xi^*,\mathbf{y}^*) = {} & \frac{1}{2}||\mathbf{w}||^2 + \\ & C\rho\sum_{i\in I_+}\xi_i + C\sum_{i\in I_-}\xi_i + \\ & C^*\rho^*\sum_{j\in J_+}\xi_j^* + C^*\sum_{j\in J_-}\xi_j^* \quad (3)\end{aligned}$$

subject to:

$$\begin{aligned} y_i(\langle \mathbf{w}, \Phi(\mathbf{x_i})\rangle_{\mathcal{H}} + b) &\geq 1-\xi_i, \quad \forall \mathbf{x_i} \in \mathbf{X_l}, i = 1,...,n_l \\ y_j^*(\langle \mathbf{w}, \Phi(\mathbf{x_j^*})\rangle_{\mathcal{H}} + b) &\geq 1-\xi_j^*, \quad \forall \mathbf{x_j^*} \in \mathbf{X_u}, j = 1,...,n_u \\ \xi_i &\geq 0, \quad \forall i = 1,...,n_l \\ \xi_j^* &\geq 0, \quad \forall j = 1,...,n_u \\ y_j^* &\in \{-1,+1\}, \quad \forall j = 1,...,n_u \end{aligned} \quad (4)$$

where C and C^* represent the cost parameters associated with margin errors induced by labeled and unlabeled training samples, respectively, and ρ and ρ^* are the cost ratios between margin errors on different sides induced by labeled and unlabeled training samples, respectively.

Our framework uses the *radial basis kernel* $k(\mathbf{x},\mathbf{x}') = e^{-\frac{||\mathbf{x}-\mathbf{x}'||^2}{\sigma}}$. Moreover, we let C^* always be C and ρ^* be a constant. Hence, we have three tunable parameters C, ρ, and σ. Given fixed C, ρ, and σ, the current implementation of our framework uses `SVM-Light` [34] to solve Equation (3), although other alternative packages such as `libsvm` [17] can also be used here. In our experiments later, we always set ρ^* to the default value given by `SVM-Light`, although in principle this parameter can also be tuned.

Next, we consider how to search for the optimal configuration, where a *configuration* means a combination of tunable parameters C, ρ, and σ in Equation (3). For a given malware classifier, its performance can be evaluated based on both its false positive rate and its detection rate, which leads to a bi-criteria optimization problem. As it is a conflicting task to minimize the false positive rate while maximizing the detection rate, we apply the Neyman-Pearson criterion [31], which seeks for a classifier that maximizes the detection rate under the constraint that its false positive rate be no greater than a certain threshold. As we do not have a closed-form formula about the detection rate and the false positive rate of a classifier based on Equation (3), we resort to cross-validation to enforce the Neyman-Pearson criterion. More specifically, we divide the training data into m folds $\{F_i\}_{i=1,2,...,m}$. Each time, we use one of the folds F_i as the test data, and the remaining F_{-i} for the purpose of training. Given any configuration, we train a classifier on F_{-i} by solving Equation (3) and test its performance on F_i. For brevity, we use $\alpha_i(C,\rho,\sigma)$ and $\beta_i(C,\rho,\sigma)$ to denote the false positive rate and the detection rate of the classifier trained from F_{-i} on testing data F_i, respectively, under configuration (C,ρ,σ).

The Neyman-Pearson criterion can be written as follows:

$$\begin{aligned} & \text{argmax}_{C,\rho,\sigma}\tilde{\beta}(C,\rho,\sigma), \quad (5) \\ \text{subject to:} \quad & \tilde{\alpha}(C,\rho,\sigma) \leq \alpha^*, \quad (6) \\ \text{where } \tilde{\beta}(C,\rho,\sigma) &= \frac{1}{m}\sum_{i=1}^{m}\beta_i(C,\rho,\sigma) \\ \tilde{\alpha}(C,\rho,\sigma) &= \frac{1}{m}\sum_{i=1}^{m}\alpha_i(C,\rho,\sigma), \end{aligned}$$

and α^* is the threshold on the average false positive rate.

In order to compare two configurations $g_1 = (C_1,\rho_1,\sigma_1)$ and $g_2 = (C_2,\rho_2,\sigma_2)$, we say that $g_1 \prec g_2$ or g_1 *outperforms* g_2 if either of the following two cases hold:

$$\begin{aligned} \text{Case 1:} \quad & \tilde{\alpha}(C_1,\rho_1,\sigma_1) \leq \alpha^* \\ & \tilde{\alpha}(C_2,\rho_2,\sigma_2) \leq \alpha^*, \text{ and} \\ & \tilde{\beta}(C_1,\rho_1,\sigma_1) > \tilde{\beta}(C_2,\rho_2,\sigma_2) \quad (7) \\ \text{Case 2:} \quad & \tilde{\alpha}(C_2,\rho_2,\sigma_2) > \alpha^* \text{ and} \\ & \tilde{\alpha}(C_1,\rho_1,\sigma_1) < \tilde{\alpha}(C_2,\rho_2,\sigma_2). \quad (8) \end{aligned}$$

We also say that $g_1 \equiv g_2$ or g_1 *is equivalent to* g_2 if $\tilde{\alpha}(C_1,\rho_1,\sigma_1) = \tilde{\alpha}(C_2,\rho_2,\sigma_2)$ and $\tilde{\beta}(C_1,\rho_1,\sigma_1) = \tilde{\beta}(C_2,\rho_2,\sigma_2)$.

Accordingly, we say that g is *the best configuration* in a configuration set if it outperforms or is equivalent to any other configuration in the set. Originally, we planned to use the coordinate descent algorithm presented in [7] to search the best configuration. However, as we are not sure how

many trials are needed for the algorithm to converge starting from any random point, the algorithm may spend a significant time on local search before exploring the entire configuration space. Hence, we decide to use the genetic algorithm, which performs local search and global search simultaneously, to find the best configuration for a given individual classifier. We treat parameters C, ρ, and σ as three types of chromosomes of the "population". In each generation, we select the k best configurations to breed the next generation. More specifically, given the current generation of the population, the next generation is reproduced as follows: **(1) Crossover:** For any two of the top k configurations, we average the values of each chromosome type. The intuition here is that the middle point between two good configurations is likely to be good. **(2) Partial mutation:** For each of the top k configurations, we replace any of its chromosomes with a randomly generated value of the corresponding type. Partial mutation explores points that differ from good configurations at only one chromosome. **(3) Full mutation:** We randomly generate values for all three types of chromosomes; full mutation prevents the search from converging to local optima. Among these three different reproduction schemes, crossover and partial mutation can be deemed as local search, as they search areas that are close to those configurations that are already found to be good; by contrast, the full mutation scheme regenerates a totally new configuration, which is used to explore good configurations in new regions at a global level.

The fitness function of a configuration $g = (C, \rho, \sigma)$ is defined as follows:

$$f(g) = \tilde{\beta}(C,\rho,\sigma) \cdot \delta(\tilde{\alpha}(C,\rho,\sigma) \leq \alpha^*) - \tilde{\alpha}(C,\rho,\sigma) \cdot \delta(\tilde{\alpha}(C,\rho,\sigma) > \alpha^*), \quad (9)$$

where $\delta(x)$ is 1 if x is true or 0 otherwise.

The algorithm is illustrated in Figure 4. The genetic algorithm has the following advantages. First, as mentioned earlier, it performs both local search and global search in each generation. This prevents the algorithm from getting stuck at local optima while improving the overall quality of the population from generation to generation. Second, both parameter k, which is used to control the number of good configurations from which to reproduce the next generation, and the number of generations that are eventually reproduced can be used to control the number of times that the search takes place. This thus provides a knob to decide how much time would be spent on searching the optimal configuration, given the computational resources available. Finally, the genetic algorithm can be easily parallelized by distributing the reproduction task over multiple processors.

5. ENSEMBLE OF CLASSIFIERS

For each type of features, we can train a classifier as discussed in Section 4. The question that naturally follows is: given the classification results from multiple individual classifiers on a new malware instance, how should we decide whether it belongs to a specific family? One widely used rule is decision by majority, that is, the same decision made by the majority of the classifiers is chosen as the final verdict. The problem with the majority rule, however, is that we have to collect all types of features that are fed to these individual classifiers, which make independent decisions on classification. This can be time consuming, as for some types

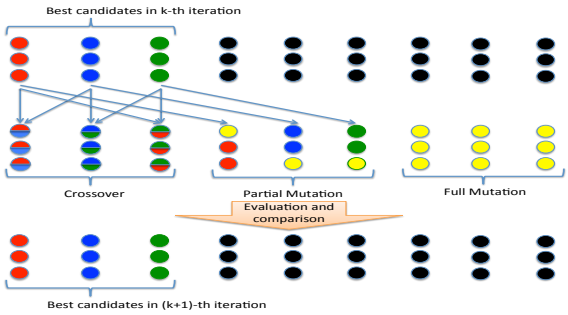


Figure 4: Illustration of genetic algorithm for searching optimal parameters

of features, it takes significant time and resources to collect their values from a new malware variant.

Due to this concern, our framework uses a simple 'OR' rule: as long as any classifier decides that a new malware instance belong to a specific family, the framework classifies it into that family. With such an 'OR' rule, our malware classification framework can draw classification results from individual classifiers in a sequential manner. As long as one classifier classifies the malware instance as positive, the framework does not need to consider the classification results by subsequent classifiers, and it is thus unnecessary to collect their corresponding types of features. In order to train the ensemble classifier as described, we enforce the *chain Neyman-Pearson criterion* on individual classifiers, which will be explained next.

5.1 Chain Neyman-Pearson Criterion

Suppose that the training data are divided into m folds $\{F_i\}_{i=1,2,...,m}$. Next, we train an individual classifier for the t-th type of features from each of $\{F_{-i}\}_{i=1,2,...m}$ and evaluate its performance on instances in F_i. Also suppose that from each malware sample we can extract T types of features. We now discuss how to modify the algorithm discussed in the previous section to train an individual classifier for the t-th type of features where $1 \leq t \leq T$.

Let $F_i^+ \subseteq F_i$ and $F_i^- \subseteq F_i$ be the set of positive and negative instances in fold F_i, respectively. Given any configuration $g = (C, \rho, \sigma)$ for the t-th type of features, let the set of positives that the candidate classifier trained from F_{-i} identifies from test data F_i be $\Theta_{t,i}(g)$. Define $\alpha^*_{1\to t}$ as:

$$\alpha^*_{1\to t} = \{\alpha_1^*, \alpha_2^*, ..., \alpha_t^*\}, \quad (12)$$

where $0 \leq \alpha_1^* \leq \alpha_2^* \leq ... \leq \alpha_t^* \leq 1$. Sequence $\alpha^*_{1\to t}$ contains false positive rate thresholds that are used to train the individual classifiers corresponding to the first t feature types.

We define $\Lambda(i,t)|_{\alpha^*_{1\to t}}$ and $\Delta(i,t)|_{\alpha^*_{1\to t}}$ recursively as in Figure 5. Hence, $\Lambda(i,t)|_{\alpha^*_{1\to t}}$ and $\Delta(i,t)|_{\alpha^*_{1\to t}}$ represent the *accumulative* set of true positives and false positives in the i-th fold, respectively, by the sequence of individual classifiers trained from the first t feature types under false positive thresholds $\alpha^*_{1\to t}$. The definitions of $\Lambda(i,t)|_{\alpha^*_{1\to t}}$ and $\Delta(i,t)|_{\alpha^*_{1\to t}}$ depend on $g_t^*|_{\alpha^*_{1\to t}}$, which, defined in Figure 6, is the best configuration found for the t-th feature type when the sequence of false positive rate thresholds used to train the individual classifiers for the first t feature types is $\alpha^*_{1\to t}$.

Hence, in contrast to the original Neyman-Pearson criterion given in Equation (5), we apply the chain Neyman-Pearson criterion: for the individual classifier trained for the t-th feature type, we search for the configuration with

$$\Lambda(i,t)|_{\alpha^*_{1\to t}} = \begin{cases} \Lambda(i,t-1)|_{\alpha^*_{1\to t-1}} \cup (\Theta_{t,i}(g^*_t|_{\alpha^*_{1\to t}}) \cap F_i^+) & t>0 \\ \emptyset & t=0 \end{cases} \tag{10}$$

$$\Delta(i,t)|_{\alpha^*_{1\to t}} = \begin{cases} \Delta(i,t-1)|_{\alpha^*_{1\to t-1}} \cup (\Theta_{t,i}(g^*_t|_{\alpha^*_{1\to t}}) \cap F_i^-) & t>0 \\ \emptyset & t=0 \end{cases} \tag{11}$$

Figure 5: Definition of $\Lambda(i,t)|_{\alpha^*_{1\to t}}$ and $\Delta(i,t)|_{\alpha^*_{1\to t}}$

$$\begin{aligned} g^*_t|_{\alpha^*_{1\to t}} &= \text{argmax}_{(C,\rho,\sigma)} \tilde{\beta}_t(C,\rho,\sigma)|_{\alpha^*_{1\to t}}, \\ \text{subject to:} \quad & \tilde{\alpha}_t(C,\rho,\sigma)|_{\alpha^*_{1\to t}} \le \alpha^*_t, \\ \text{with } \tilde{\beta}_t(C,\rho,\sigma)|_{\alpha^*_{1\to t}} &= \frac{1}{m}\sum_{i=1}^{m} \frac{|(\Theta_{t,i}(C,\rho,\sigma) \cap F_i^+) \cup \Lambda(i,t-1)|_{\alpha^*_{1\to t}}|}{|F_i^+|}, \\ \tilde{\alpha}_t(C,\rho,\sigma)|_{\alpha^*_{1\to t}} &= \frac{1}{m}\sum_{i=1}^{m} \frac{|(\Theta_{t,i}(C,\rho,\sigma) \cap F_i^-) \cup \Delta(i,t-1)|_{\alpha^*_{1\to t}}|}{|F_i^-|}, \end{aligned}$$

Figure 6: Definition of $g^*_t|_{\alpha^*_{1\to t}}$

the highest accumulative detection rate over the first t feature types under the constraint that the accumulative false positive rate over the first t feature types is no higher than α^*_t. With redefined $\tilde{\alpha}_t(C,\rho,\sigma)$ and $\tilde{\beta}_t(C,\rho,\sigma)$, we keep the same method of comparing two configurations as described in Section 4. Hence, given the current false positive rate threshold α^*_t, the algorithm presented in Section 4 can be used to train each individual classifier sequentially based on the chain Neyman-Pearson criterion.

5.2 Dynamic Programming

The application of the chain Neyman-Pearson criterion requires us to know the sequence of false positive thresholds, $\{\alpha^*_1, \alpha^*_2, ..., \alpha^*_T\}$, where T gives the total number of feature types in the classification framework. Given that the overall false positive rate threshold is α^*, we then should have $\alpha^*_T = \alpha^*$. Clearly, how to set these false positive rate thresholds affects the performance of the ensemble of classifiers trained based on the chain Neyman-Pearson criterion. In an extreme case, if we let α^*_1 be α^* and we can find a classifier based on the first feature type that has a false positive rate equal to α^*, then for the remaining feature types, the classifiers trained based on the chain Neyman-Pearson should not lead to any false positives in order for the ensemble of classifiers to have a false positive rate no greater than α^*.

Searching the continuous space of $\{\alpha^*_1, \alpha^*_2, ..., \alpha^*_T\}$ for an optimal configuration can be computationally prohibitive, as these parameters are dependent on each other. Moreover, during the search process, we want to tease out those features that contribute little to the accumulative classification performance, which is typically done in a separate feature selection process. To overcome these challenges, we propose a dynamic programming method, as illustrated in Figure 7, to search an optimal setting of $\alpha^*_{1\to T}$. Consider the following set of false positive rate thresholds that should be satisfied by individual classifiers,

$$\{0\alpha^*/D, 1\alpha^*/D, 2\alpha^*/D, ..., D\alpha^*/D\}, \tag{13}$$

where D is a predefined parameter, and we let the false positive rate thresholds in the sequence $\{\alpha^*_1, \alpha^*_2, ..., \alpha^*_T\}$ take values only from this set. For brevity, we define $\mathcal{D} = \{0, 1, ..., D\}$.

Threshold	Feature 1	...	Feature j-1	Feature j	...	Feature T
0α*/D	g*$_1$(0α*/D)		g*$_{j-1}$(0α*/D)	g*$_j$(0α*/D)		X
1α*/D	g*$_1$(1α*/D)		g*$_{j-1}$(1α*/D)	g*$_j$(1α*/D)		X
...	...		...	...		X
(i-1)α*/D	g*$_1$((i-1)α*/D)		g*$_{j-1}$((i-1)α*/D)	g*$_j$((i-1)α*/D)		X
iα*/D	g*$_1$(iα*/D)		g*$_{j-1}$(iα*/D)	**g*$_j$(iα*/D)**		X
...	...		...	...		X
α*	g*$_1$(α*)		g*$_{j-1}$(α*)	g*$_j$(α*)		g*$_T$(α*)

Figure 7: Dynamic programming for searching optimal configurations. $g^*_j(i\alpha^*/D)$, which is the best configuration when the false positive rate threshold over all the first j features is $i\alpha^*/D$, is obtained based on the best configurations in the previous column, $g^*_j(k\alpha^*/D)$ where $k = 0, 1, ..., i$. For the last feature (i.e., feature T), only the configuration under false positive rate threshold α^* needs to be computed.

Consider the t-th feature type, where $1 \le t \le T$, and any $d \in \mathcal{D}$. Given that the false positive rate threshold for the t-th feature type under the chain Neyman-Pearson criterion is $d\alpha^*/D$, we use the equations in Figure 8 to find recursively the optimal threshold for the false positive rate of the $(t-1)$-th feature type, which is $prev(t,d) \cdot \alpha^*/D$. The crux here is that when we search the optimal value for $prev(t,d)$, we check all possible false positive rate thresholds for the previous feature type (note that that previous threshold should be no greater than the current threshold), and use the one that leads to the best configuration for the current feature type under the chain Neyman-Pearson criterion. When we calculate the best configuration $g^*_t|_{d,d'}$ when the current and the last false positive rate thresholds are $d\alpha^*/D$ and $d'\alpha^*/D$, respectively, we keep track of both $\Lambda(i,t,d)$ and $\Delta(i,t,d)$, which represent the set of true positives and false positives, respectively, under the condition that the current false positive threshold is $d\alpha^*/D$ and the previous false positive thresholds are set to the values that recursively lead to the best configuration for the current feature type t.

The calculation can be done in an approach based on dynamic programming. For the first feature type (i.e., $t = 1$),

$$
\begin{aligned}
prev(t,d) &= \begin{cases} \text{argmax}_{d' \in \mathcal{D}} \quad g_t^*|_{d,d'} & t > 1 \\ -1 & t = 1 \end{cases} \\
\text{where} \quad g_t^*|_{d,d'} &= \text{argmax}_{(C,\rho,\sigma)} \tilde{\beta}_t(C,\rho,\sigma)|_{d'}, \\
\text{subject to:} \quad & \tilde{\alpha}_t(C,\rho,\sigma)|_{d'} \leq d\alpha^*/D, \\
\text{with } \tilde{\beta}_t(C,\rho,\sigma)|_{d'} &= \frac{1}{m}\sum_{i=1}^{m} \frac{|(\Theta_{t,i}(C,\rho,\sigma) \cap F_i^+) \cup \Lambda(i,t-1,d')|}{|F_i^+|}, \\
\tilde{\alpha}_t(C,\rho,\sigma)|_{d'} &= \frac{1}{m}\sum_{i=1}^{m} \frac{|(\Theta_{t,i}(C,\rho,\sigma) \cap F_i^-) \cup \Delta(i,t-1,d')|}{|F_i^-|},
\end{aligned}
$$

Figure 8: Definition of $prev(t,d)$ and $g_t^*|_{d,d'}$

$$
\begin{aligned}
\Lambda(i,t,d) &= \begin{cases} \Lambda(i,t-1,prev(t,d)) \cup (\Theta_{t,i}(g_t^*|_{d,prev(t,d)}) \cap F_i^+) & t > 0 \\ \emptyset & t = 0 \end{cases} \\
\Delta(i,t,d) &= \begin{cases} \Delta(i,t-1,prev(t,d)) \cup (\Theta_{t,i}(g_t^*|_{d,prev(t,d)}) \cap F_i^-) & t > 0 \\ \emptyset & t = 0 \end{cases}
\end{aligned}
$$

Figure 9: Definition of $\Lambda(i,t,d)$ and $\Delta(i,t,d)$

we calculate the best configuration for each possible false positive rate threshold $d\alpha^*/D$, where $d \in \mathcal{D}$. Once the best configurations for the $(t-1)$-th feature type have been found, we search the best configuration for the t-th feature type for each possible false positive rate threshold $d\alpha^*/D$, where $d \in \mathcal{D}$. For any such d, we check the d possible false positive rate thresholds for the previous feature type (i.e., $0, 1\alpha^*/D, 2\alpha^*/D, ..., d\alpha^*/D$), and assuming that the previous classifier uses its best configuration for each of these false positive rate thresholds, we look for the best configuration for the current feature type.

It is noted that for the last feature type, i.e., the T-th feature type, we only need to consider the false positive rate threshold $D\alpha^*/D$. Define an *operation* to be an execution of the genetic algorithm, which is used to solve Equation (12). We have the following proposition:

Proposition 3. *If the dynamic programming algorithm performs as described, it takes* $2(D+1) + (T-2) \cdot \frac{(D+1)(D+2)}{2}$ *operations to calculate the best configurations.*

Proof. For the first feature type, the algorithm performs $D+1$ operations that calculate the best configurations, each for a threshold $d\alpha^*/D$ with $d \in \mathcal{D}$. For the last feature type (i.e., the T-th feature type), the best configurations are only searched under the false positive rate threshold $D\alpha^*/D$; as the previous threshold for the $(T-1)$-th feature type can be any of $d\alpha^*/D$ with $0 \leq d \leq D$, $D+1$ operations that calculate the best configurations are performed. For the k-th feature type where $2 \leq k \leq T-1$, when the current false positive rate threshold is $d\alpha^*/D$, d operations that calculate the best configurations are performed (note that the previous threshold should be no greater than $d\alpha^*/D$). Hence, for each of these feature types, $\sum_{d=0}^{D} d = (D+1)(D+2)/2$ operations are performed. In total, there are thus $2(D+1) + (T-2) \cdot \frac{(D+1)(D+2)}{2}$ operations used to calculate the best configurations.

Implementation. If exhaustive search for optimal configurations were allowed, the detection rates with the configurations in the matrix in Figure 7 would be (weakly) monotonically improving from left top to right bottom. Due to a limited number of tries when searching for an optimal configuration with the genetic algorithm, it is however possible that no feasible configuration can be found or the configuration found has a false positive rate not falling into the current threshold range. In such circumstances, extra efforts are needed to ensure that classification performances of the entries in the matrix are indeed monotonically improving.

Let matrix M contain the configurations found; that is to say, $M(d,t)$ gives the configuration (α, β) found when the threshold on the false positive rate is $d\alpha^*/D$ for the t-th feature type; if no configuration is found, $M(d,t)$ is `null`. To optimize the process of finding the best configurations, we update the t-th column in two passes when the search for the t-th feature type is finished:

First pass: Consider the following example. The three thresholds on false positive rates are 0.01, 0.02, and 0.03. Suppose that the corresponding configurations found are (0.005, 0.8), (0.001, 0.99), and (0.025, 0.9). Hence, both the first two configurations have a false positive rate falling in the range [0, 0.01]. In this case, we use the second one to overrule the first one because it leads to a higher detection rate. Hence, in the first pass, for the configuration found for the d-th threshold, we proceed as follows:

(a.1) *Its false positive rate is higher than* $d\alpha^*/D$. As this violates the Neyman-Pearson criterion, we let $M(d,t)$ be `null`.

(a.2) *Its false positive rate is no greater than* $d\alpha^*/D$ *but higher than* $(d-1)\alpha^*/D$. In this case, we keep the current solution found for this threshold.

(a.3) *Its false positive rate is no greater than* $(d-1)\alpha^*/D$. Let the false positive rate of the solution found be r. We calculate $d' = \lceil rD/\alpha^* \rceil$ and compare the configuration previously found under the false positive rate threshold $d'\alpha^*/D$. If the current solution outperforms the previous configuration, we replace the previous configuration under the false positive rate threshold $d'\alpha^*/D$ with the current solution, and set the current solution under the false positive rate threshold $d\alpha^*/D$ to be `null`.

Second pass: Using the same example, after the first pass, the three configurations become (0.001, 0.99), `null`, and $(0.025, 0.9)$. Note that albeit having a higher false positive rate, the third configuration has a lower detection rate, suggesting that it is dominated by the first configuration. Hence, in the second pass, we ensure that all configurations in the matrix M are ordered by the performances. Let $\max_\beta M_{i,j}$ be the maximum detection rate among the first i rows and j columns of matrix M. Without loss of generality, if $M_{i,j}$ is `null`, its detection rate is assumed to be 0.

For the d-th configuration where $1 \leq d \leq D$, let its detection rate be $\beta_{d,t}$. We have the following cases.

(b.1) $\max_\beta M_{d,t-1} \geq \max_\beta M_{d-1,t} \geq \beta_{d,t}$. This means that classification from the current feature type does not help improve the detection accuracy. Hence, we *inherit* the classification results from the previous feature type with the same false positive rate threshold without using the current feature type.

(b.2) $\max_\beta M_{d,t-1} \geq \max_\beta M_{d-1,t} \geq \beta_{d,t}$. This implies that the current feature type helps but increasing the false positive rate threshold does not help. We thus *deactivate* the configurations found from the current threshold for this feature type and will not consider any further results based on $M(d,t)$. We say that $M(d,t)$ is now *inactive*.

(b.3) $\beta_{d,t} > \max\{\max_\beta M_{d,t-1}, \max_\beta M_{d-1,t}\}$. This means that both the current threshold and the feature type help improving the detection accuracy, and we thus keep its classification results.

After the two passes, we ensure that performances of the configurations in matrix M, if they are active, must increase (weakly) monotonically as we increase the false positive rate threshold or add more feature types. These two passes are performed immediately for each feature type once the configurations are learned based on the chain Neyman-Pearson criterion under all the false positive rate thresholds. Due to the optimization applied in our implementation, the count stated in Proposition 3 provides only an upper bound on the real number of operations that have been executed.

It is noted that the overhead in feature extraction varies with the type of the features. When we train the ensemble classifier under the chain Neyman-Pearson criterion, feature groups are ordered based on their relative extraction overhead in an non-decreasing manner. Hence, costly features are used only if the cheap ones do not have sufficient discriminative power in classifying the malware family being considered.

6. EXPERIMENTS

We use those unpacked samples in the malware dataset described in Section 2 for performance evaluation, although in practice packed samples can be unpacked first before extracting features from them. Due to limited resources available, we extract features only for those samples that have been labeled. In total we have 15,494 labeled unpacked samples for experiments, and their family breakdown is as follows: `Bagle` (152), `Bifrose` (1677), `Hupigon` (4748), `Koobface` (371), `Ldpinch` (190), `Lmir` (181), `Rbot` (923), `Sdbot` (253), `Swizzor` (1276), `Vundo` (2852), `Zbot` (1231), and `Zlob` (2140). A classifier is built for each of these families, based on the one-against-all rule, which treats samples from the family under study as positive and all others as negative.

Among all the samples, we randomly choose 80% of them for training, and the remaining 20% for testing. In the training dataset, 75% of them are marked as unlabeled. When using the chain Neyman-Pearson criterion to train classifiers, we further divide the training dataset into five folds (i.e., $m = 5$ in Section 5.1), four of them used to search the best configurations and the remaining one used to evaluate the performance of a configuration. It is noted that during the training phase based on cross validation, the 20% of the test data are not available. Our task is to train a cost-sensitive classifier from the training dataset, and then label every instance in the test dataset. We set the false positive rate allowed for the ensemble classifier to be at most 5% (i.e., $\alpha^* = 0.05$). For the purpose of dynamic programming, we partition the overall false positive rate into 6 intervals, i.e., $D = 6$ in Equation (13).

Moreover, when using the genetic algorithm to search the optimal configuration, we generate candidate solutions for three generations, among which solutions in the first generation are randomly generated based on the full mutation scheme (see Figure 4). From the population of each generation, we choose the top three configurations for further reproduction. Hence, three new configurations are generated from crossover, and three others from partial mutation. In addition, three more are produced from full mutation. The size of the total population per generation is thus 9.

The six types of features mentioned in Section 2 are considered, `PE-num`, `PE-bool`, `Hexdump 2-gram`, `Objdump 1-gram`, `PIN 2-gram`, and `PIN SysCall`. We use the logistic regression method discussed in [40] to choose 100 features from each of the six feature types. When we search optimal configurations for each feature type, we order the six feature types based on the relative difficulty of obtaining their values. As both PE header features and `Hexdump 2-gram` features can be obtained in a deterministic manner, we consider them during the early phase of the classifier ensemble. `Objdump 1-gram` features require disassembly code of the executable programs, which can be obtained through static analysis. PIN features require dynamic execution of the programs, and are thus more difficult to obtain. Hence, we train individual classifiers sequentially according to the following order of the six feature types: `PE-num`, `PE-bool`, `Hexdump 2-gram`, `Objdump 1-gram`, `PIN 2-gram`, and `PIN SysCall`.

6.1 Classification Performance

Table 1 presents the average performances of the configurations found by our method over the five test folds in terms of false positive rates and detection rates. Clearly, for any of these malware families, it is not necessary to include all six feature types. For instance, for the `Swizzor`, `Zbot`, and `Zlob` families, only the features extracted from the PE headers are needed to classify their instances. This is desirable because for some feature types such as those from dynamic execution, it takes significant efforts to collect their feature values. Hence, our method based on the chain Neyman-Pearson criterion has the effect of selecting only those feature types that are useful for automated malware classification. Moreover, it is observed that classification of different malware families requires different types of features. For instance, `Hexdump 2-gram` features are only useful for the `Ldpinch` and the `Lmir` families but not the other ones. This is plausible

Table 1: Performances trained under the chain Neyman-Pearson criterion. The numbers are the averages of 5-fold cross validation. '-' means that the feature type is not used. False positive rate threshold is 5%.

Family	PE-num	PE-bool	Hexdump 2-gram	Objdump 1-gram	PIN 2-gram	PIN SysCall
Bagle	(0.34%, 91.3%)	(1.1%, 98.0%)	-	(1.6%, 100.0%)	-	-
Bifrose	(0.7%, 81.7%)	-	-	(2.0%, 91.2%)	(2.2%, 94.2%)	(2.5%, 95.8%)
Hupigon	(1.1%, 91.5%)	(1.9%, 98.4%)	-	-	-	(3.7%, 99.4%)
Koobface	(0.06%, 90.9%)	(0.09%, 94.6%)	-	(0.96%, 100%)	-	-
Ldpinch	(0.4%, 46.7%)	(1.9%, 83.1%)	(3.04%, 90.5%)	-	-	-
Lmir	(0.35%, 70.7%)	(0.4%, 92.8%)	(1.96%, 95%)	-	-	-
Rbot	(2.8%, 85.1%)	(3.2%, 91.5%)	-	-	-	(3.67%, 92.6%)
Sdbot	(1.2%, 58.3%)	(1.5%, 73.7%)	-	-	-	(1.6%, 76.5%)
Swizzor	(0.33%, 97.9%)	(0.5%, 99.7%)	-	-	-	-
Vundo	(0.33%, 95.6%)	(1.0%, 99.3%)	-	-	-	-
Zbot	(0.76%, 87.9%)	(1.6%, 92.2%)	-	-	(3.6%, 93.4%)	(4.0%, 96.0%)
Zlob	(1.5%, 98.6%)	(2.1%, 100.0%)	-	-	-	-

as the uniqueness of a malware family may manifest itself over only a specific subset of feature types. Our algorithm is able to find these distinguishing feature types.

Figure 10 shows the classification performance for each malware family. From Figure 10(1), we observe that the false positive rate for classifying the test samples in each malware family is no greater than the predefined threshold of 5% for each malware family, suggesting that the classifier ensemble trained is indeed able to enforce the Neyman-Pearson criterion. The detection rate per family is shown in Figure 10(2). For most of the families, we observe that the detection rate is above 90%, implying that the majority of the samples of these families can be identified by our method. For the `Sdbot` family, however, the detection rate is only around 70%. We note that for this family, during the training phase, our algorithm cannot find good configurations to achieve high classification accuracy anyway: the combination of PE header features and the `PIN-SysCall` features can only lead to an average detection rate of 76.5% based on 5-fold cross validation. Hence, the poor detection rate of `Sdbot` samples attributes to the six feature types used here which are unable to distinguish samples in the `Sdbot` family as effectively as those in the other ones. Comparing the detection rates in Figure 10(2) against those in Table 1, we find that the former are typically lower than the corresponding values by the classifier ensembles seen in Table 1. This is feasible as the detection rates in Figure 10(2) are obtained on the test data, which are invisible to the training phase; by contrast, during the training phase, we search configurations that lead to the highest detection rates under the false positive rate constraint using 5-fold cross validation, which are shown in Table 1.

For a classifier, let the number of true positives, false positives, true negatives, and false negatives be n_{tp}, n_{fp}, n_{tn} and n_{fn}, respectively. Its *precision* is $n_{tp}/(n_{tp}+n_{fp})$, and its *recall* $n_{tp}/(n_{tp}+n_{fn})$. The F-1 score, the harmonic mean of precision and recall, is $2n_{tp}/(2n_{tp}+n_{fp}+n_{fn})$. The precision, recall, and F-1 score of the classifier trained for each of the 12 malware families are shown in Figure 10(3). We note that the recall measures are high for most of the malware families (except the `Sdbot` family), because the recall measure is essentially the same as the detection rate. However, the precision measure can be low for some of the families, such as `Bagle`, `Ldpinch`, `Lmir`, and `Sdbot`. Common to these families is the fact that they are severely underrepresented, having a much smaller number of instances than the other families. Consider any malware family, whose samples comprise a fraction p of the entire dataset. Let the false positive rate and the detection rate of the classifier trained for this family be α and β, respectively. Then, we have:

$$precision = \frac{p\beta}{(1-p)\alpha + p\beta} \leq \frac{1}{1+\alpha(1/p-1)}. \quad (14)$$

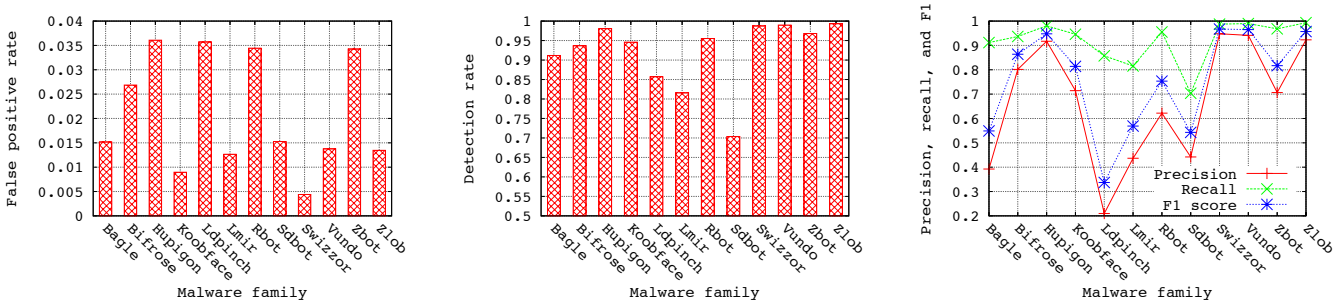
The inequality holds as $\beta \leq 1$, so if p is small, meaning that the malware family is significantly underrepresented, then $1/p-1$ is large, and precision becomes small even though all positive samples can be detected successfully.

In some cases, however, it is important to have a classifier with high precision. For instance, if we need to study common characteristics shared by samples by a malware family, such as its unique string signatures, it is desirable to have a classifier that produces only a small fraction of false positives among the samples labeled as positive. For such cases, there are two solutions. First, we can use a smaller false positive rate in Equation (14), which can cancel the effect of $1/p-1$. For instance, we use a smaller false positive rate threshold 0.5%, and the classification performances for the `Ldpinch` and the `Lmir` families are shown as follows:

Measure	Ldpinch	Lmir
False positive rate	0.063%	0.16%
Detection rate	75.0%	88.6%
Precision	0.923	0.861
Recall	0.750	0.886
F1 Score	0.828	0.873

With a lower false positive rate threshold, the classifier trained by our framework reduces the false positive rate, and thus leads to a higher precision measure. Under this lower false positive rate threshold, the majority of the samples labeled as positive by the classifiers indeed belong to the malware families under study. For the `Lmir` family, although a false positive rate threshold of 0.5% is used, the detection rate is even higher than what we have observed when the false positive rate threshold is 5%. This results from the fluctuations due to randomness in searching for the optimal configurations by the genetic algorithm and the randomness in partitioning the dataset for training and testing.

An alternative way of training a classifier ensemble with a high precision measure is to slightly modify the chain Neyman-Pearson criterion: rather than ensuring that the accumulative false positive rate is no greater than a certain threshold, we search for configurations with the highest detection rates under the constraint that the precision measure should be no smaller than a predefined threshold. If such a threshold is set to be, say, 0.8, we expect the classifier to produce positive labels among which at least 80% of them are true positives.



(1) False positive rate (2) Detection rate (3) F1 score

Figure 10: Performances of the classifiers trained under the chain Neyman-Pearson criterion

6.2 Performance Comparison

In the previous subsection, we have demonstrated the classification performance using the chain Neyman-Pearson criterion. Next we show how it compares against existing ensemble learning techniques. For the sake of performance comparison, we consider the following methods. (1) *Imputed*: We combine all the features used in the previous subsection together, and use the means of available values to impute those missing ones. We further use three standard classifiers, kNN, SVM, and decision trees (C4.5) to train classifiers from a subset of the imputed dataset. (2) *Boosting*: Using each of the three classifiers from (1) as the base learner, we further use the boosting method as the ensemble classifier. (3) *Bagging*: Using each of the three classifiers from (1) as the base learner, we further use the bagging method as the ensemble classifier. (4) *Stacking*: We stack all the three classifiers from (1) as the base learners together to create an ensemble classifier. Due to space limitation, we omit the details of these methods here and refer interested readers to the existing literature (e.g., [29, 8, 24]). For fair comparison, we use the implementations of these methods from a third-party machine learning software, `Orange` [23]. In order for other researchers to reproduce our results, we use the default settings in `Orange` in all the experiments, except that for the stacking scheme, we use kNN rather than Naive Bayes as the meta learner. Using the latter leads to very poor performance for malware classification (not shown here due to space limitation), and this has also been observed in our earlier work [40].

In a new set of experiments, 80% of samples are labeled and used to train the classifier and the remaining 20% are used for testing. Note that in the experiments, all the samples in the training dataset are labeled. The classification results are shown in Table 2. It is observed that none of the standard approaches is able to produce false positive rates below 5% for all the families: for each of these schemes, at least five families have false positive rates higher than 5%. Even with more labeled samples for training, the number of malware families with a detection rate below 80% varies from one to four among existing approaches, the best comparable to our proposed scheme. These results suggest that the ensemble classifier trained based upon the chain Neyman-Pearson criterion outperforms the existing methods.

The superior performance of our scheme is explained as follows. First, the implementation by Orange, the software we use for the standard methods, is not necessarily optimal. Second, our method aggressively searches the best parameter settings of the individual classifiers under the Neyman-Pearson criterion, but the standard solutions do not. Last but not least, our approach seeks to optimize the collective performance of the entire classifier ensemble in a sequential manner, which is not done by the existing methods.

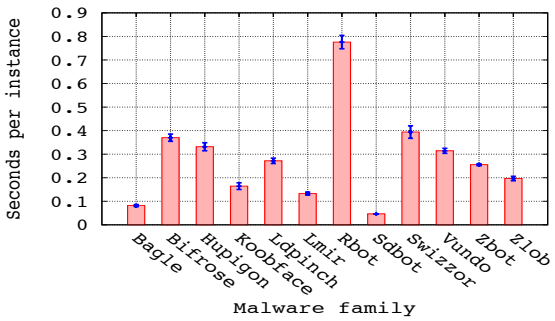


Figure 11: Execution time per instance

6.3 Execution Time for Online Classification

As our goal is to automate the process of malware classification, one may wonder whether our proposed malware classification framework is suitable for online malware classification. We thus perform another set of experiments to examine the average execution time spent on each malware instance in the test dataset. We run our method on a Linux workstation with 12 1.6Hz cores and 64G memory.

In our experiments, we only consider the execution time spent on malware classification. Hence, the time spent on collecting the features that are necessary for classification is *not* counted in the execution time we show here. The test dataset contains 3199 malware instances, which is fed to each of the 12 classifier ensembles we have trained in Table 1. For each classifier ensemble, we perform malware classification over the entire test dataset for 20 times. The mean execution time per family is shown in Figure 11, together with its standard deviation over these 20 runs.

Over all the 12 classifier ensembles, the mean execution time per malware instance is merely 0.2779 second, which suggests that when we apply our malware classification framework on a new malware variant to test whether it belongs to a specific family, the majority of the time would be spent on collecting necessary features that are used for classification, rather than the classification process itself. However, if we want to test whether a new malware variant belongs to any of existing families, we would have to test it against each of the classifier ensembles trained for these families. Still, even though we have to test the malware variant against the ensemble classifiers for 100 families, the total execution time

Table 2: Performance comparison. The experiments use 80% of labeled samples for training, and the other 20% for testing. All numbers are in percentage. False positive rates (FPR) higher than 5% and detection rates (DR) below 80% are shown in bold.

Family	Imputed						Boosting						Bagging						Stacking	
	KNN		Tree		SVM		KNN		Tree		SVM		KNN		Tree		SVM			
	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR
Bagle	**13.2**	**60.6**	**10.8**	89.7	**6.7**	**9.0**	**9.1**	84.0	**10.8**	89.7	**6.7**	**9.0**	**12.0**	**61.3**	**8.0**	89.2	3.5	**22.1**	**8.1**	86.0
Bifrose	**6.5**	90.6	**5.3**	95.0	**8.0**	94.7	**6.4**	93.9	**5.3**	95.0	**8.0**	94.7	**6.7**	90.7	4.3	97.0	**8.5**	94.4	4.4	95.6
Hupigon	2.0	97.4	1.1	98.9	2.6	97.8	1.0	97.7	1.1	98.9	2.6	97.8	2.1	97.4	0.9	99.2	2.5	97.8	0.9	99.1
Koobface	**6.8**	98.6	3.7	97.9	1.2	98.1	**5.6**	94.1	3.7	97.9	1.2	98.1	**6.3**	98.1	3.0	97.8	1.4	98.2	2.6	97.3
Ldpinch	**17.9**	**65.9**	**15.4**	**79.6**	**19.3**	**74.2**	**12.5**	**69.2**	**15.4**	**79.6**	**19.3**	**74.2**	**14.9**	**65.1**	**5.6**	81.3	**15.3**	**72.2**	**9.9**	81.8
Lmir	**7.6**	**52.2**	**15.8**	82.8	**8.8**	80.7	**10.4**	80.1	**15.8**	82.8	**8.8**	80.7	**8.4**	**50.1**	**7.1**	87.2	**7.4**	80.1	**10.4**	84.7
Rbot	**31.8**	90.0	**7.4**	93.9	**23.3**	93.2	**25.8**	91.5	**7.4**	93.9	**23.3**	93.2	**32.5**	92.1	**7.3**	95.8	**21.8**	95.8	**6.6**	95.2
Sdbot	**18.3**	**36.7**	**23.9**	**73.2**	**22.4**	**45.8**	**33.6**	**50.3**	**23.9**	**73.2**	**22.4**	**45.8**	**9.5**	**36.2**	**18.2**	**75.2**	**8.0**	**46.0**	**20.0**	**74.9**
Swizzor	3.9	97.9	0.9	99.0	2.8	96.9	1.4	98.3	0.9	99.0	2.8	96.9	3.3	98.0	0.5	99.4	3.5	96.8	1.0	99.6
Vundo	3.1	97.6	1.7	98.4	1.4	98.9	1.3	98.1	1.7	98.4	1.4	98.9	3.1	97.7	1.2	99.2	1.4	98.9	1.0	99.2
Zbot	2.8	87.6	4.2	95.4	3.7	95.0	3.7	91.3	4.2	95.4	3.7	95.0	1.8	87.5	3.0	97.3	3.4	95.0	3.8	96.3
Zlob	**7.8**	98.3	1.4	98.5	3.0	98.2	4.1	98.5	1.4	98.5	3.0	98.2	**7.4**	98.3	0.7	99.0	2.9	98.3	1.9	99.0

is still expected to be less than half an minute. This time can be further reduced if multiple cores are used.

Interestingly, there are significant differences among the execution times by the classifier ensembles trained for the 12 malware families. On average, the classifier ensemble trained for the `Rbot` family takes the longest time, which is 0.776 second, and the one trained for the `Sdbot` family takes the least time, which is 0.0466 second. The differences can be attributed to a number of reasons, such as the set of features used for classification, fraction of positive samples, how likely positive samples can be detected at the early stage of the classifier ensemble, and the execution times of `SVM-Light` under various configurations.

6.4 Analysis of Genetic Algorithm

We next show how the genetic algorithm performs in finding the optimal configurations. We examine the executions of the genetic algorithm over all the 12 malware families, and find that about 50.8% of them do not return plausible solutions, which lead to unused feature types seen in Table 1 (i.e., those marked with '-'). Next we consider only the cases in which the genetic algorithm returns a plausible solution.

Generation	Fraction	Crossover	Partial Mutation	Full Mutation
1	49.2%	0	0	100%
2	24.6%	22.2%	35.6%	42.2%
3	26.2%	27.1%	39.6%	33.3%

The above table breaks down the fraction of solutions found from each generation. It is noted that almost half of the configurations found come from the first generation, which essentially uses full mutation to randomly generate solutions. Each of the second and third generations produces one fourth of the plausible solutions. One interesting observation is that among the solutions in the second and third generations, the full mutation scheme still contributes to a significant fraction of these solutions; however, when the reproduction process continues, the fraction of solutions randomly generated from full mutation decreases. This is because the evolutionary process gradually improves the quality of the population from both the crossover and the partial mutation schemes. Hence, if there are only a small number of searches used to find the optimal configuration, global search, which randomly explores the entire configuration space by full mutation, is more efficient in finding a good solution; however, with more searches allowed, local search, either crossover or partial mutation, is able to focus the searches in those regions with good solutions and can thus improve the current solutions more efficiently. This suggests that the genetic algorithm can combine the advantages of both local search and global search in finding optimal configurations.

7. RELATED WORK AND DISCUSSIONS

Due to scalability advantages over manual malware analysis, machine learning has been applied in a number of previous studies for distinguishing malware programs from benign ones (*e.g.*, [30, 14, 25, 27, 1, 32]). Malware detection is a different task from malware classification, which is the theme of this study, as the goal of the later is to classify malware variants into their corresponding families. In some recent works, machine learning has also been applied to automate the process of malware classification (*e.g.*, [21, 40, 15, 19, 38, 20, 39]). The main differences among these works lie in the types of features used for malware classification. Although the malware features used in this study are far from being exhaustive, they cover key malware features collected from both static analysis and dynamic analysis, and those features considered in the previous efforts can be easily incorporated in our malware classification framework based on the chain Neyman-Pearson criterion. More importantly, what really distinguishes our work from these previous efforts is that they *do not* consider different requirements on different types of errors induced by malware classification. For instance, if we want to study the trend of a malware family, we expect that the majority of the samples in this family should indeed belong to this family, which requires us to have a malware classifier with a low false positive rate even though we have to sacrifice its detection rate. Hence, it is of important to have the flexibility of balancing the false positive rate and the detection rate in malware classification, which cannot be achieved by any of existing methods.

The performance evaluation of of this work relies upon a malware dataset with labels derived from consensus from AV software detection results. It is known that the labels obtained in such a way may lead to a biased dataset [16, 18]. Even with such a flaw, it is noted that using the same dataset, the proposed solution outperforms standard ensemble learning techniques in classification performances. In the future, we plan to use other malware datasets with truthful labels to further evaluate our approach.

Like many other machine learning techniques, the proposed ensemble classifier is trained assuming that the test data would have the same or a similar distribution as the training data. However, the population of malware variants in a malware family can be nonstationary, which is called concept drift in parlance of machine learning [33]. In an adversarial environment, concept drift can bring significant challenges to malware classification [13]. Concept drift

requires us to monitor changes of malware populations; if an abrupt change is observed, the classifiers need to be retrained. Adapting the proposed ensemble malware classifier to deal with concept drift remains as our future work.

In this work, we use the cost-sensitive SVM as the basic block to train individual classifiers. Davenport *et al.* considered how to tune the costs in cost-sensitive SVM based on the Neyman-Pearson criterion [7] with a coordinate descent method. The genetic algorithm used in this work can be easily parallelized, and has a single parameter (i.e., the number of generations) to control the number of search attempts.

The existing ensemble learning techniques, such as boosting, bagging, and stacking [9], can be orthogonal to our work: they can be used to improve the performance of an individual classifier when a specific feature type is considered in our malware classification framework. Moreover, the way in which our work combines multiple classifiers also differs from these existing techniques. First, the 'OR' rule used in our method enables us to take advantage of the correlation among multiple feature types: If adding a new feature type does not help improve the classification performance over previous feature types, our classification framework does not collect its values from a malware sample. Second, missing feature values render it difficult to apply existing methods directly. For example, given a malware sample, if we cannot collect values for a specific feature type, we cannot assign a weight to it in the boosting algorithm. Third, by applying the chain Neyman-Pearson criterion, we recursively train a set of classifiers that leads to the optimal performance collectively. Here, the performance of a classifier ensemble is evaluated according to the Neyman-Pearson criterion, rather than the classification errors used in some existing methods.

Acknowledgment We thank anonymous reviewers for their comments and our paper shepherd, Aziz Mohaisen, for his great help on improving the final version of this paper.

8. REFERENCES

[1] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *Journal of Computer Virology*, 7(4):247–258, 2011.

[2] `http://securitywatch.pcmag.com/security/323419-symantec-says-antivirus-is-dead-world-rolls-eyes`.

[3] E. B. Baum and D. Haussler. What size net gives valid generalization? *Neural computation*, 1(1):151–160, 1989.

[4] `http://cnx.org/content/m11548/1.2/`.

[5] C. J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[6] O. Chapelle, B. Schölkopf, and A. Zien. *Semi-supervised learning*, volume 2. MIT press Cambridge, 2006.

[7] M. A. Davenport, R. G. Baraniuk, and C. D. Scott. Tuning support vector machines for minimax and neyman-pearson classification. *IEEE Trans. Pattern Anal. Mach. Intell.*, 32(10):1888–1898, October 2010.

[8] T. G. Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*. Springer, 2000.

[9] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[10] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Trans. on Knowledge and Data Engineering*, 2009.

[11] `http://www.imperva.com/docs/HII_Assessing_the_Effectiveness_of_Antivirus_Solutions.pdf`.

[12] `http://www.pintool.org/`.

[13] A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In *ACM AISec'13*.

[14] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Maching Learning Research*, 7:2721–2744, December 2006.

[15] D. Kong and G. Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of ACM KDD'13*, 2013.

[16] P. Li, L. Liu, D. Gao, and M. K Reiter. On challenges in evaluating malware clustering. In *RAID'10*.

[17] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

[18] A. Mohaisen and O. Alrawi. AV-meter: An evaluation of antivirus scans and labels. In *Proceedings of DIMVA'14*.

[19] A. Mohaisen and O. Alrawi. Unveiling zeus: automated classification of malware samples. In *Proceedings of the 22nd international conference on WWW companion*, 2013.

[20] A. Mohaisen, A. G. West, A. Mankin, and O. Alrawi. Chatter: Exploring classification of malware based on the order of events. In *IEEE Conference on Communications and Network Security (CNS'14)*.

[21] L. Nataraj, V. Yegneswaran, P. Porras, and J. Zhang. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Proceedings of ACM AISec'11*.

[22] `http://www.offensivecomputing.net/`.

[23] `http://orange.biolab.si/`.

[24] N. C. Oza and K. Tumer. Classifier ensembles: Select real-world applications. *Information Fusion*, 9(1), 2008.

[25] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *ACSAC'08*.

[26] K. Raman. Selecting features to classify malware. In *Proceedings of InfoSec Southwest*, 2012.

[27] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4):639–668, December 2011.

[28] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *IEEE Symposium on Security and Privacy*, pages 65–79. IEEE, 2012.

[29] M. Saar-Tsechansky and F. Provost. Handling missing values when applying classification models. *Journal of Machine Learning Research*, 8:1623–1657, December 2007.

[30] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, 2001.

[31] C. Scott and R. Nowak. A neyman-pearson approach to statistical learning. *IEEE Transactions on Information Theory*, 51(11), 2005.

[32] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *RAID'09*.

[33] A. Singh, A. Walenstein, and A. Lakhotia. Tracking concept drift in malware families. In *Proceedings of ACM AISec'12*.

[34] `http://svmlight.joachims.org/`.

[35] Symantec Internet security threat report. Symantec Corporation, 2011.

[36] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[37] `https://www.virustotal.com/`.

[38] A. G. West and A. Mohaisen. Metadata-driven threat classification of network endpoints appearing in malware. In *Proceedings of DIMVA'14*.

[39] Z. Xu, J. Zhang, G. Gu, and Z. Lin. Autovac: Towards automatically extracting system resource constraints and generating vaccines for malware immunization.

[40] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'13)*. 2013.