Discriminant Malware Distance Learning on Structural Information for Automated Malware Classification

Deguang Kong[*]
Dept. of Computer Science and Engineering
University of Texas at Arlington
doogkong@gmail.com

Guanhua Yan
Information Sciences Group (CCS-3)
Los Alamos National Laboratory[†]
ghyan@lanl.gov

ABSTRACT

The voluminous malware variants that appear in the Internet have posed severe threats to its security. In this work, we explore techniques that can automatically classify malware variants into their corresponding families. We present a generic framework that extracts structural information from malware programs as attributed function call graphs, in which rich malware features are encoded as attributes at the function level. Our framework further learns discriminant malware distance metrics that evaluate the similarity between the attributed function call graphs of two malware programs. To combine various types of malware attributes, our method adaptively learns the confidence level associated with the classification capability of each attribute type and then adopts an ensemble of classifiers for automated malware classification. We evaluate our approach with a number of Windows-based malware instances belonging to 11 families, and experimental results show that our automated malware classification method is able to achieve high classification accuracy.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning; D.4.6 [**Operating System**]: Security and Protection

Keywords

Malware, distance learning, metric learning, structure, graph matching, optimization, function call graph

1. INTRODUCTION

Malware are responsible for a large number of malicious activities in the cyber space, such as spamming, identity theft, and DDoS (Distributed Denial of Service) attacks. Behind the sheer number of malware instances, however, lies the fact that a large number of them came from the same origins. More than 75 percent of malware detected belong to as few as 25 families, based on the 2006 Microsoft Security Intelligence report [15]. For the instances belonging to the same malware family, we can study their common characteristics and develop defensive methods accordingly, much alike developing vaccines against a specific flu family (*e.g.*, swine flu). Accurate prediction of the evolution trend of a malware family also enables us to deploy effective mitigation methods in advance and thus alleviate the damage caused by this malware family.

A question that naturally follows is: *how should we classify a large number of malware instances into their corresponding families*? Anti-Virus (AV) companies commonly rely on signatures, such as strings and regular expressions, to determine malware families, but it is well known that signature-based methods are error-prone and can be easily evaded by intelligent malware programs. On the other hand, manually reverse-engineering every malware variant to figure out its lineage requires advanced skills and is often a time-consuming, sometimes even tedious, process.

Therefore, there is an urgent need of developing methods that can automatically classify malware instances into their corresponding families accurately. To achieve automated malware classification, we need to extract useful information – or *features* in parlance of machine learning – from labeled samples for which we know their families, and build a model that predicts which family a newly observed malware instance belongs to based on the feature values it carries. Although it sounds a standard supervised learning procedure, we are faced with a fundamental challenge when constructing malware features: the rich structural information contained in malware programs, such as their function call graphs and basic block graphs, is not amenable to traditional supervised learning techniques, which usually operate on numerical vectorial representations of data objects.

Against this backdrop, the goal of this work is to develop a framework that automatically classifies malware instances according to their inherent rich structural information. This framework extracts the function call graph from each malware program, and collects various types of fine-grained features at the function level, such as what system calls are made and how many I/O read and write operations have been made in each function. For each type of features, our framework evaluates the similarity of two malware programs by iteratively applying the following two basic techniques: (1) *discriminant distance metric learning*, which projects the original feature space into a new one such that malware instances belonging to the same family are closely clustered while clusters formed by different malware families are separated with large margins; (2) *pairwise graph matching*, which aims to find the right pairwise function-level matching between the function call graphs of two malware instances in order to measure their structural similarity. The similarity score estimated between two malware instances for each type of features reflects the likelihood that they

[*]This work is done when Deguang Kong was working at Los Alamos National Laboratory.

[†]Los Alamos National Laboratory Publication No. LA-UR 12-26699

should be classified into the same malware family – if observed feature values of that type are used as our *evidence*. We further learn our *confidence* level in each type of evidence and henceforth build a classifier that predicts the family of a new malware instance by combining different types of evidences with their corresponding confidence levels.

In a nutshell, our key contributions are summarized as follows. **(1)** We present a generic framework that extracts structural information from a malware program and represents it as an attributed function call graph, where fine-grained malware features are encoded as attributes associated with each function node in the graph. **(2)** We formulate the processes of discriminant distance metric learning and pairwise graph matching as two optimization problems, and develop novel eigen-based methods to solve them. **(3)** Our framework adaptively learns the confidence levels associated with different types of evidences provided to the ensemble of classifiers by assigning increasingly higher penalty to those training samples misclassified previously. **(4)** With extensive experiments, we demonstrate that our proposed method is able to classify malware instances into their corresponding families with high accuracy.

The remainder of this paper is organized as follows. Section 2 states the problem to be addressed in this work, and Section 3 describes the overview of our methodology. Section 4 discusses how to extract features based on function call graphs. We further present our method for malware distance learning in Section 5 and how to use an ensemble of classifiers for automated malware classification in Section 6. Section 7 shows experimental results. We present related work in Section 8 and draw concluding remarks in Section 9.

2. PROBLEM STATEMENT

In this work, we are interested in the problem of classifying malware instances into their corresponding families automatically. Let Y be the set of different malware families. To start with, we have a labeled dataset with n_l elements, $L = \{(x_1, y_1), (x_2, y_2), ..., (x_{n_l}, y_{n_l})\}$ where x_i is a malware instance and $y_i \in Y$ is the family that malware x_i belongs to, for $1 \leq i \leq n_l$. The labeled dataset can include those samples manually labeled by malware experts who reverse-engineered the malware programs, or be obtained through consensus by major AV software. Our goal is to develop a model or classifier that can accurately predicts the family of an unseen malware sample $f : X \rightarrow Y$, where X denotes the set of all possible unseen malware samples. It is noted here the classifier f we aim to build only consider known malware families, and we are thus not interested in identifying new malware families.

In order to build a classifier f, we first need to extract useful information from each labeled malware instance x_i. Feature extraction from malware programs can be done through either *static analysis* or *dynamic analysis*. Static analysis refers to studying a malware's code statically without actually executing it, and by contrast, dynamic analysis runs the malware program (usually in a virtual controlled environment) and understands its run-time behavior. Although dynamic analysis has the advantage of revealing the true behavior of often obfuscated malware programs, it requires a virtual execution environment, which makes it more demanding than static analysis. Hence, this study focuses on features extracted from only static analysis.

Static malware features considered in the literature include byte sequence n-gram [21, 11, 18], disassembly code [2], and PE header fields [23, 19]. These features, however, do not embody the rich structural information inherent in malware programs. The function call graph obtained from disassembly analysis, for instance, represents the calling relationships among functions, and thus reflects the overall structure of the malware program. Compared with the aforementioned types of static features, *structural information is more difficult to obfuscate*, and can thus be used as robust features for classifying malware instances.

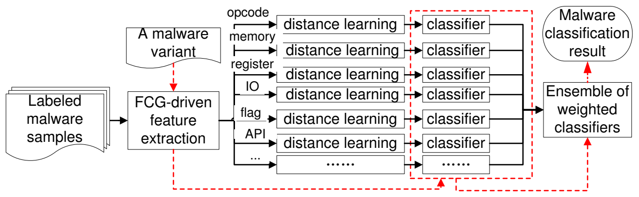


Figure 1: Overview of our automated malware classification framework (solid lines are used for the training process, and dashed line for the process of classifying a new malware variant)

Such structural information, however, poses significant technical challenges, as it is not amenable to standard supervised learning methods, which usually operate on numerical vectorial representations of data objects. In order to apply automated malware classification on structural information inherent in malware programs, it is necessary to solve the following problems: **(1)** How to extract and represent structural information from malware programs? **(2)** How to effectively compute the distance between two malware instances given their structural information? **(3)** How to build an automated malware classifier based on distance measures among malware programs? Our work offers a framework that tackles these three problems in a *principled* way, as illustrated in the following sections.

3. OVERVIEW OF METHODOLOGY

The overview of our automated malware classification framework is depicted in Figure 1. The training phase includes the following four key steps.

Step 1: FCG-driven feature extraction. To extract structural information from a malware program, we first disassemble the malware program, and build its function call graph. The function call graph is further used to drive the process of feature extraction: for every node (i.e., a function) in the graph, we extract various types of *attribute*, including what library APIs are made and how many I/O read and write operations have been made in this function. Information regarding each type of features is represented as a vector of numerical values. For example, for library API attribute, each element in the vector provides the number of times a corresponding API has been called in this function. After Step 1, each labeled malware program is abstracted into an attributed function call graph, where each function node contains a number of feature vectors.

Step 2: Discriminant malware distance learning. The next step concerns how to compute the distance between two malware distances represented as their attributed function call graphs. For each type of attribute, we project the original feature space onto a new one such that malware instances belonging to the same family are closely clustered while clusters formed by different malware families are separated with large margins. Moreover, we perform pairwise graph matching, which aims to find the right pairwise function-level matching between the attributed function call graphs of two malware instances for the purpose of measuring their structural similarity.

Step 3: Training individual classifiers. For each type of features, once we have computed the similarity between any two labeled malware instances, we train an individual classifier for it. Our framework is open to any classifier that, in order to classify a new sample, requires only information of a set of *anchor instances*, which are usually the subset of labeled samples in the original dataset. Such classifiers include the kNN classifier, for which the anchor instance set includes the k closest instances from the test instance, and the SVM classifier, whose support vector contains all the anchor instances.

Step 4: Building ensemble of weighted classifiers. For each type of features we have considered, the similarity measure between two malware instances reflects the likelihood that they belong to the same family. Given a new malware variant, for each type of features, we form its *evidence* as the distance it is from each of its anchor instances as well as the label information of each anchor instance. The *type* of an evidence is defined to be the type of *attribute* from which it is formed. For different types of *attribute*, we can have different confidence levels about their evidences, because some attribute types are more indicative of a malware's lineage than the others. To learn the confidence level associated with a type of evidence, we use an Adaboost-like approach, which gives an increasingly higher penalty to training samples that are wrongly classified. We henceforth build a classifier that predicts the family of a new malware instance by combining different types of evidences according to their corresponding confidence levels.

The output of the training phase of our automated malware classification framework is an ensemble of classifiers. Given a new unknown malware sample, we first construct its function call graph from the disassembly code, and for each function node in it, we extract different types of attribute. Next, for each type of attribute, we form its evidence that describes the distance between the new sample and the anchor instances as well as how each anchor instance is labeled. We then feed the evidence to the corresponding individual classifier. By combining all the evidences, the ensemble of weighted classifiers makes the final decision on which malware family it should be classified into.

4. FCG-DRIVEN FEATURE EXTRACTION

Structural information inherent in a malware program can be represented at two different resolutions: *functions* and *basic blocks*. The function call graph of a program, which is a directed graph, represents the calling relationships among the functions. A basic block in a program is a piece of code with a single entry point and a single exit point, and the transitions among basic blocks form the control flow graph of the program. In this work, we use function call graph (FCG for short) to drive the process of extracting important features from malware programs.

The FCG captures the calling relationship of a program, and each vertex in it represents a local function. For each local function, we first translate it into an intermediate language and then extract six types of attributes from it. They include `opcode` (the frequency of appearances for each opcode), `API` (the number of times each library API function is called), `memory` (the number of memory reading and writing operations made in this function), `IO` (the number of I/O reading and writing operations), `Register` (the number of reading and writing operations on each register), and `Flag` (the number of changes on each flag). For each attribute type, we represent it as a feature vector associated with the local function.

The output of the step of FCG-driven feature extraction is an *attributed FCG*, which contains an FCG with vertices each carrying a number of feature vectors, as illustrated in Figure 2. An example of extracting features over FCG is shown in the following, where this attributed FCG abstracts our knowledge about a malware program.

```
A function in malware:
-----------------------------------
sub_42765E proc near;
1: mov ebx,[esi+4h]
2: cmp ebx, 12h
3: jnz loc_17871211
```

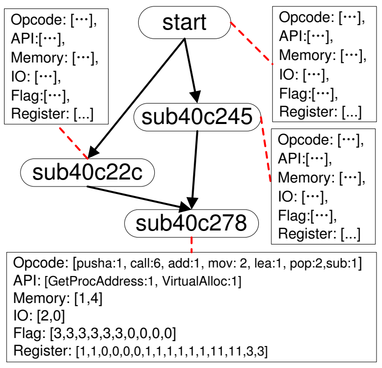


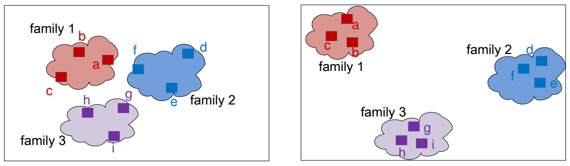
Figure 2: Illustration of an attributed FCG

Table 1: Notations used in the paper

notation	description
n_l	number of labeled malware
n_c	number of malware families
C_k	set of instances belonging to malware the kth family
$\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i)$	attributed FCG of malware i
$\mathcal{F}^q_{im}$	feature vector of attribute q at node m in malware i's attributed FCG
$D^q_{i,j}$	pairwise malware distance between $\mathcal{G}_i$ and $\mathcal{G}_j$ w.r.t attribute type q
S^q_w	malware within-class distance (scalar) w.r.t attribute type q
S^q_b	malware between-class distance (scalar) w.r.t attribute type q
$\mathbf{A}^{ij}$	$\in \Re^{\lvert\mathcal{V}^i\rvert\times\lvert\mathcal{V}^j\rvert}$, pairwise malware FCG function matching matrix between $\mathcal{G}_i$ and $\mathcal{G}_j$
$\mathbf{A}^{ij}_{mn}$	is 1 if function m in malware $\mathcal{G}_i$ matches function n in $\mathcal{G}_j$, and is 0 otherwise
$\mathbf{W}^q$	$\Re^{d_q\times d}$, distance metric learning matrix for attribute type q
$\mathbf{U}_w$	$\in \Re^{d\times d}$ within-class operator
$\mathbf{U}_b$	$\in \Re^{d\times d}$ between-class operator
$D_{im,jn}$	node distance between function m in $\mathcal{G}_i$ and function n in $\mathcal{G}_j$
$D_{im,jn;im',jn'}$	edge distance between node pairs (m, m') in $\mathcal{G}_i$ and (n, n') in $\mathcal{G}_j$
p	$= \lvert\mathcal{V}^i\rvert \times \lvert\mathcal{V}^j\rvert$: total number of pairwise function matching between $\mathcal{G}_i$ and $\mathcal{G}_j$
$\mathcal{M}$	$\in \Re^{p\times p}$: pairwise graph matching matrix between $\mathcal{G}_i$ and $\mathcal{G}_j$
$\mathbf{a}$	$\in \Re^p$: graph matching vector between $\mathcal{G}_i$ and $\mathcal{G}_j$
$\mathbf{a}_t$	$= 1$ if node $m$in $\mathcal{G}_i$ is matched with node n in $\mathcal{G}_j$; $=$ 0 otherwise, where $m = ((t-1)/\lvert\mathcal{V}_j\rvert + 1), n = \mathrm{mod}\ ((t-1), \lvert\mathcal{V}_j\rvert) + 1$.
$\mathbf{Z}$	$\in \Re^{n_l\times Q}$, $\mathbf{Z}_{iq} = 1$ if malware i is correctly labeled w.r.t. attribute type q; 0 otherwise.
α	$\in \Re^{Q\times 1}$, confidence level for attribute type q

```
4: mov eax, [edx]
5: call ds:SwitchDesktop
sub_42765E endp;
-----------------------------------
extraction of features:
-----------------------------------
Attribute 1: Opcode: [mov:2, and:1, call: 1, cmp:1, jnz:1]
Attribute 2: API: [ds:SwitchDesktop:1]
Attribute 3: Memory: [MemoryR: 3,  MemoryW: 1]
Attribute 4: IO: [IOR: 0, IOW: 0]
Attribute 5: Flag: [CF: 2, AF: 1, OF: 2, PF: 2, SF: 2,
ZF: 2, TF: 0, IF: 0, RF: 0, DF: 0]
Attribute 6: Register: [EAXR: 0, EAXW: 2, EBXR: 0, EBXW: 1,
ECXR: 0, ECXW: 0, EDXR: 1, EDXW: 0,
ESIR: 1, ESIW: 0, EDIR: 0, EDIW: 0,
ESPR: 1, ESPW: 1, EBPR: 0, EBPW: 0].
-----------------------------------
```

In the next section, we shall discuss how to evaluate the similarity of two malware programs based on their attributed FCGs. As a number of notations will be used in the next few sections, we summarize all these notations in Table 1 for clarity.



(1) Before learning (2) After learning

Figure 3: Demonstration of malware distance metric learning

5. DISTANCE METRIC LEARNING

Automated malware classification requires methods to evaluate the distances among malware instances. Having extracted an attributed FCG graph for each malware program, we next look for the appropriate distance metric to compute the distance between two malware programs based on attributed FCG graphs.

5.1 Maximum margin principle

Our search for malware distance metric is guided by the *maximum margin principle*, i.e., the malware in the same family should be closely clustered while clusters formed by different malware families should have large margins to separate them. See Fig. 3 for a motivating example. Before distance metric learning, malware from the same class may have a large distance than those from different classes based on the naive Euclidean distance (e.g., `dist(a,b)` $>$ `dist(a,f)` in Fig. 3(1)). After learning the appropriate distance metric, which projects data points in the original feature space into a different space, the intra-class distance is decreased while the inter-class distance is increased. Following the same example, we have `dist(a,f)>>dist(a,b)`, as shown in Fig. 3(2).

More formally, let $\mathcal{G}_i$ be the attributed function call graph of malware i where $1 \leq i \leq n_l$, and E_q be the feature vector of type q where $q \in T$. Let $D^q_{i,j}$ represent the pairwise distance between malware $\mathcal{G}_i$ and $\mathcal{G}_j$ computed according to attribute type q. Then, the *within-class distance* S^q_w, which is the sum of squared distances among all pairs of malware belonging to the same family according to feature type q, is given by:

$$S^q_w = \sum_k \sum_{i \in C_k} \sum_{j \in C_k} (D^q_{i,j})^2. \tag{1}$$

Similarly, the *between-class distance* S^q_b, which is the sum of squared distances among all pairs of malware belonging to different families according to attribute type q, is given by:

$$S^q_b = \sum_k \sum_{l, l \neq k} \sum_{i \in C_k} \sum_{j \in C_l} (D^q_{i,j})^2, \tag{2}$$

Thus following the maximum margin principle, we need to maximize the between-class distance while minimizing the within-class distance for attribute type q, i.e.,

$$\min(S^q_w - S^q_b). \tag{3}$$

Note we use a trace difference criterion in Eq. (3). As is known in the machine learning community, it has some advantages over optimizing the trace quotient criterion (i.e., $\max \frac{S^q_b}{S^q_w}$), such as convexity formulation, ease for manipulation, etc. To minimize Eq. (3), we need to compute pairwise malware distance D^q_{ij} for attribute type q, which will be shown next.

5.2 Malware pairwise distance computation

Given $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i)$ and $\mathcal{G}_j(\mathcal{V}_j, \mathcal{E}_j)$, the attributed FCGs of two malware samples i and j, their distance D_{ij} can be computed through the graph distance between $\mathcal{G}_i$ and $\mathcal{G}_j$. To simplify computation, we assume there exists one-to-one match between function nodes in $\mathcal{G}_i$ and $\mathcal{G}_j$. Letting $\mathcal{V}^m_i$ be the mth function node in malware $\mathcal{G}_i$, the pairwise graph distance problem breaks down into the following two subproblems: **(1)** How to compute the optimal pairwise node match matrix $\mathbf{A}^{ij} \in \Re^{|\mathcal{V}_i| \times |\mathcal{V}_j|}$ for malware $\mathcal{G}_i$ and $\mathcal{G}_j$, and **(2)** How to compute the optimal pairwise node distance between $\mathcal{V}^m_i$ and $\mathcal{V}^n_j$.

For problem (1), we learn the pairwise malware graph matching matrix $\mathbf{A}^{ij}$, where $\mathbf{A}^{ij}_{mn} = 1$ denote node m in malware $\mathcal{G}_i$ matches node n in malware $\mathcal{G}_j$, and $A^{ij}_{mn} = 0$ otherwise. We call problem (1) the *graph match* **A**-*learning problem.* [1] For problem (2), supposing that $\mathcal{F}^q_{im}$, the feature vector of type q at node m in malware i's attributed FCG, is a d_q-dimension vector (note that d_q varies with the feature type q), our goal is to learn distance metric $\mathbf{W}^q \in \Re^{d_q \times d}$, where d is the dimension of the new subspace after projection, **w.r.t. each attribute type** q,[2]:

$$D_{im,jn} = \sqrt{(\mathcal{F}_{im} - \mathcal{F}_{jn})^T \mathbf{W}\mathbf{W}^T (\mathcal{F}_{im} - \mathcal{F}_{jn})} \tag{4}$$

For example, if $\mathbf{W} = \mathbf{I}$, distance metric D is Euclidean distance. Actually, the learned distance metric $\Sigma = \mathbf{W}\mathbf{W}^T, \Sigma \in \Re^{d_q \times d_q}$ is a symmetric semi-definite positive matrix, and now metric D is Mahalanobis distance. Note here $\mathbf{W}$ is required to be orthonormal, i.e., $\mathbf{W}^T\mathbf{W} = \mathbf{I}$. We call problem (2) the *distance metric* **W**-*learning problem*. This process is done for each attribute type q.

Once we have obtained distance metric $\mathbf{W}$ and pairwise graph matching matrix $\mathbf{A}$, we define the pairwise malware distance $D_{i,j}$ for attribute type q (for simplicity, we ignore superscript q here) as:

$$D_{i,j} = \sqrt{\frac{\sum_{mn} D^2_{im,jn} \mathbf{A}^{ij}_{mn}}{|\mathcal{V}_i||\mathcal{V}_j|} + \frac{\sum_{mn,m'n'} \mathbf{A}^{ij}_{mn} \mathbf{A}^{ij}_{m'n'} D^2_{im,jn;im',jn'}}{|\mathcal{E}_i||\mathcal{E}_j|}} \tag{5}$$

where $D_{im,jn}$ is the node distance between node m in $\mathcal{G}^i$ and node n in $\mathcal{G}^j$ as defined in Eq. (4), $\mathbf{A}^{ij}_{mn}$ represents unary assignment (*node match*) for matching of node m in $\mathcal{G}^i$ with node n in $\mathcal{G}^j$; and the second term inside the square root represents pairwise assignment (*edge match*), where both node $m \in \mathcal{V}^i$ matches with $n \in \mathcal{V}^j$, and also node $m' \in \mathcal{V}^i$ matches with $n' \in \mathcal{V}^j$. That is, not only the substitution of nodes but also the edge structure play a role in the computation of distance between the attributed FCGs of two malware programs (more details will be introduced in §6.4).

A key observation is that the distance between two attributed FCGs contributed by edges mirrors the number of edge deletion/insertion operations, and thus if the graph matching is given, this portion of distance is fixed (see §6.4 for more details). Define J_{ij} as follows:

$$J_{i,j} = \frac{\sum_{mn,m'n'} \mathbf{A}^{ij}_{mn} \mathbf{A}^{ij}_{m'n'} D^2_{im,jn;im',jn'}}{|\mathcal{E}_i||\mathcal{E}_j|}. \tag{6}$$

Then we have:

$$\begin{aligned} D^2_{i,j} &= \sum_{m \in \mathcal{V}_i} \sum_{n \in \mathcal{V}_j} D^2_{im,jn} + J_{i,j} \\ &= \frac{\sum_{m \in \mathcal{V}_i} \sum_{n \in \mathcal{V}_j} \mathbf{A}^{ij}_{mn} (\mathcal{F}_{im} - \mathcal{F}_{jn})^T \mathbf{W}\mathbf{W}^T (\mathcal{F}_{im} - \mathcal{F}_{jn})}{|\mathcal{V}_i||\mathcal{V}_j|} + J_{i,j}. \end{aligned} \tag{7}$$

Then substituting Eq. (7) into Eqs. (1, 2), we have:

$$\begin{aligned} S_w &= \mathrm{Tr}(\mathbf{W}^T \mathbf{U}_w \mathbf{W}) + \sum_k \sum_{i \in C_k} \sum_{j \in C_k} J_{i,j}, \\ S_b &= \mathrm{Tr}(\mathbf{W}^T \mathbf{U}_b \mathbf{W}) + \sum_k \sum_{l, l \neq k} \sum_{i \in C_k} \sum_{j \in C_l} J_{i,j}. \end{aligned} \tag{8}$$

[1] We use one-to-one match, for two malware $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$ and $\mathcal{G}_j = (\mathcal{V}_j, \mathcal{E}_j)$, they may have different number of function nodes, e.g., $|\mathcal{V}_i| > |\mathcal{V}_j|$, $(|\mathcal{V}_i| - |\mathcal{V}_j|)$ virtual nodes (with null evidence values) are added to make a alignment.

[2] For clarity, we ignore the superscript q here.

where

$$\mathbf{U}_w = \sum_k \sum_{i \in C_k} \sum_{j \in C_k} \sum_{m \in \mathcal{V}_i} \sum_{n \in \mathcal{V}_j} \frac{\mathbf{A}^{ij}_{mn}(\mathcal{F}_{im} - \mathcal{F}_{jn})(\mathcal{F}_{im} - \mathcal{F}_{jn})^T}{|\mathcal{V}_i||\mathcal{V}_j|},$$

$$\mathbf{U}_b = \sum_k \sum_{l, l \neq k} \sum_{i \in C_k} \sum_{j \in C_l} \sum_{m \in \mathcal{V}_i} \sum_{n \in \mathcal{V}_j} \frac{\mathbf{A}^{ij}_{mn}(\mathcal{F}_{im} - \mathcal{F}_{jn})(\mathcal{F}_{im} - \mathcal{F}_{jn})^T}{|\mathcal{V}_i||\mathcal{V}_j|}.$$

Substituting Eqs. (1, 2, 7) into the optimization goal in Eq. (3), we have:

$$\min_{\mathbf{W}} \text{Tr}(\mathbf{W}^T\mathbf{U}_w\mathbf{W} - \mathbf{W}^T\mathbf{U}_b\mathbf{W}) + \lambda_q B, \quad s.t. \quad \mathbf{W}^T\mathbf{W} = \mathbf{I} \quad (9)$$

for each attribute type q, where

$$B = \sum_k \sum_{i \in C_k} \sum_{j \in C_k} J_{i,j} - \sum_k \sum_{l, l \neq k} \sum_{i \in C_k} \sum_{j \in C_l} J_{i,j} = const,$$

and λ_q is the coefficient for the cost contributed by B associated with edge cost. As once the graph matching $\mathbf{A}$ is fixed, B is a constant and we thus need to solve the following optimization problem in each iteration:

$$\min_{\mathbf{W}} \text{Tr}(\mathbf{W}^T\mathbf{U}_w\mathbf{W} - \mathbf{W}^T\mathbf{U}_b\mathbf{W}), \quad s.t. \quad \mathbf{W}^T\mathbf{W} = \mathbf{I} \quad (10)$$

Note in the above formulation, we need to solve: (1) distance metric $\mathbf{W}$ for each attribute type q, (2) graph matching $\mathbf{A}$ during computation of $\mathbf{U}_w$ and $\mathbf{U}_b$ for pairwise malware distance.

Next, we present a learning framework to solve the above problem of Eq. (10). The key idea is to apply the expectation maximization algorithm [5], to iteratively update parameters $\mathbf{W}$ and $\mathbf{A}$. More specifically, we repeatedly perform the following two steps: (1) estimate pairwise malware matching matrix $\mathbf{A}$, given the current distance metric $\mathbf{W}$ (§5.3); (2) predict the optimal distance metric $\mathbf{W}$, given the current pairwise malware matching matrix $\mathbf{A}$ (§5.4). These two steps are iterated for several iterations. To summarize, the whole algorithm is illustrated in Alg. 1.

5.3 W-learning algorithm

In Eq. (10), given the current graph matching $\mathbf{A}$, the optimal solution $\mathbf{W}$ can be obtained according to the following theorem:

THEOREM 1. *[Ky Fan [26]] Let $\mathcal{H} \in \Re^{d\times d}$ be a symmetric matrix with eigenvalues $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_d$ and the corresponding eigenvectors $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_d]$, then*

$$\min_{\mathbf{W}^T\mathbf{W}=\mathbf{I}_k} Tr(\mathbf{W}^T\mathcal{H}\mathbf{W}) = \sum_{i=1}^{k} \lambda_i. \quad (11)$$

Moreover, the optimal $\mathbf{W}^* = [\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_k]$ subject to orthonormal transformation.

To apply Ky Fan's theorem here, we first let $\mathcal{H} = (\mathbf{U}_w - \mathbf{U}_b)$. We can thus obtain the optimal solution $\mathbf{W}$ with d smallest eigenvectors of $\mathcal{H}$.

5.4 Pairwise graph matching A-learning algorithm

We now discuss how to match nodes in the attributed FCGs of two malware programs. Note that the unary distance $D_{im,jn}$ is determined with Eq. (4), which represents the node distance; for pairwise distance $D_{im,jn;im',jn'}$, it can be determined in four different cases, which are illustrated in Fig. 4, according to the graph structure of $\mathcal{G}_i$ and $\mathcal{G}_j$.

(1) There exist both an edge from node m to m' in $\mathcal{G}_i$ and also an edge from n to n' in $\mathcal{G}_j$ (Fig. 4(a)). It means the edge $e_{mm'} \in \mathcal{E}_i$ is substituted with edge $e_{nn'} \in \mathcal{E}_j$. Since both pairwise nodes (m, m') and (n, n') have been matched in the unary node matching process, no edit cost is needed for transforming one edge in $\mathcal{G}_i$ to another in $\mathcal{G}_j$, i.e., $D_{im,jn;im',jn'} = 0$;

(2) Only edge (m, m') exists, or only edge (n, n') edge exists (Fig. 4(b) or 4(c)). It means one edge insertion/deletion operation is needed. We assume all the edge insertion/deletion operation has the same cost, and thus assign $D_{im,jn;im',jn'} =$ const.

(3) Neither edge (m, m') nor edge (n, n') exist in $\mathcal{G}_i$ or $\mathcal{G}_j$ (Fig. 4(d)). No edge edit operation is needed to transform edge (m, m') to (n, n'), and thus $D_{im,jn;im',jn'} = 0$.

In order to perform pairwise graph matching to obtain $\mathbf{A}$[1], we first establish the following lemma:

LEMMA 1. *Let $p = |\mathcal{V}_i| \times |\mathcal{V}_j|$, $\mathbf{a} \in \Re^p$, $r = (m-1) \times |\mathcal{V}_j| + n$, and $s = (m'-1) \times |\mathcal{V}_j| + n'$[2]. Finding $\mathbf{A}^{ij}$ to minimize the distance between $\mathcal{G}_i$ and $\mathcal{G}_j$, i.e.,*

$$\min_{\mathbf{A}^{ij}} D_{i,j}$$

as shown in Eq. (5), is equivalent to solving:

$$\min_{\mathbf{a}} f(\mathbf{a}) = \mathbf{a}^T\mathcal{M}\mathbf{a} \quad s.t. \quad \mathbf{a}_i = \{1, 0\}, \quad (12)$$

where $\mathcal{M} \in \Re^{p\times p}$ encodes the distance metric computed from $D_{im,jn}$ and $D_{im,jn;im',jn'}$. That is to say, diagonals of matrix $\mathcal{M}_{r,r} = \frac{D_{im,jn}}{|\mathcal{V}_i||\mathcal{V}_j|}$ is computed from Eq. (4), and off-diagonals of matrix $\mathcal{M}_{r,s} = \frac{D_{im,jn;im',jn'}}{|\mathcal{E}_i||\mathcal{E}_j|}$ where $s \neq r$ is computed according to the four cases discussed before.

PROOF. First note that finding the solution to $\min_{\mathbf{A}^{ij}} D_{i,j}$ is equivalent to solving:

$$\min_{\mathbf{A}^{ij}} D^2_{i,j} \quad (13)$$

[1]For computational cost consideration, we need not actually add virtual node in the computation of Eqs.(12,17). After we get $(\min |\mathcal{V}_i|, |\mathcal{V}_j|)$ pairwise one-to-one match, we just match the unmatched $(||\mathcal{V}_i| - |\mathcal{V}_j||)$ nodes to virtual nodes with null values.

[2]Here we use "row-first" order to assign the order.

Algorithm 1 distance metric learning algorithm

```
Input: (1) Attributed FCGs of labeled malware programs; (2) T: Maximum number of iterations
Output: The learned distance metric $\mathbf{W}^q$ for each attribute type $q$
Procedure:
1:  Initialization: distance metric $\mathbf{W}^q = \mathbf{I}_{d_q}$, iteration $t = 1$
2:  for $t = 1, 2, \cdots$, T do
3:      for each pairwise malware $(\mathcal{G}_i, \mathcal{G}_j)$ do
4:          Update graph matching $\mathbf{A}$ using Eq. (17) given current distance metric $\mathbf{W}$
5:          Compute pairwise distance $D^q_{i,j}$ of Eq. (7)
6:      end for
7:      for each attribute type $q$ do
8:          Compute within-class and between-class distances $\mathbf{U}^q_w$ and $\mathbf{U}^q_b$ in Eq. (9)
9:      end for
10:     Update distance metric $\mathbf{W}$ with Eq. (11)
11:     $t = t + 1$
12: end for
```

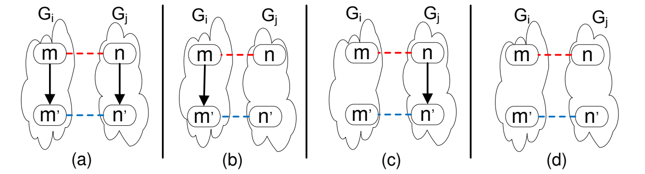


Figure 4: Four cases for pairwise distance computation of $D_{im,jn;im',jn'}$.

Supposing $\mathbf{a}_r = \mathbf{A}^{ij}_{mn}$ and $\mathbf{a}_s = \mathbf{A}^{ij}_{m'n'}$, we now establish the relationship between Eq. (13) and Eq. (12). Clearly,

$$\begin{aligned}\mathbf{a}^T\mathcal{M}\mathbf{a} = \sum_{rs}\mathbf{a}_r\mathcal{M}_{rs}\mathbf{a}_s = \sum_r \mathbf{a}_r^2\mathcal{M}_{rr} + \sum_r\sum_{s,s\neq r}\mathbf{a}_r\mathcal{M}_{rs}\mathbf{a}_s \\ = \sum_r \mathbf{a}_r\mathcal{M}_{rr} + \sum_r\sum_{s,s\neq r}\mathbf{a}_r\mathcal{M}_{rs}\mathbf{a}_s.\end{aligned} \tag{14}$$

In Eq. (5), for the node distance,

$$\frac{\sum_{mn} D_{im,jn}\mathbf{A}^{ij}_{mn}}{|\mathcal{V}_i||\mathcal{V}_j|} = \sum_r \mathbf{a}_r\mathcal{M}_{r,r} \tag{15}$$

for the edge distance,

$$\frac{\sum_{mn,m'n'}\mathbf{A}^{ij}_{mn}\mathbf{A}^{ij}_{m'n'}D_{im,jn;im',jn'}}{|\mathcal{E}_i||\mathcal{E}_j|} = \sum_{r,s}\mathbf{a}_r\mathbf{a}_s\mathcal{M}_{r,s} \tag{16}$$

By making a sum over of Eq. (15) and Eq. (16) on both sides, we have LHS = RHS, where LHS gives Eq. (13) and RHS gives Eq. (12). This completes the proof. $\square$

Optimal solution: Eq. (12) is a discrete optimization problem, which is NP-hard and thus hard to handle. We relax the constraint $\mathbf{a}_i \in \{1, 0\}$ to $\mathbf{a}^T\mathbf{a} = 1$, and have:

$$\min_{\mathbf{b}} f(\mathbf{b}) = \mathbf{b}^T\mathcal{M}\mathbf{b} \quad s.t. \quad \mathbf{b}^T\mathbf{b} = 1. \tag{17}$$

Clearly, according to Theorem 1 (Ky Fan [26]), we obtain the optimal solution of $\mathbf{b}$ is given by the smallest eigenvector of $\mathcal{M}$. We use $\mathbf{b}$ to obtain the pairwise graph matching $\mathbf{A}$ as follows.

Computation of pairwise graph matching $\mathbf{A}$: Before computing pairwise graph matching matrix $\mathbf{A}^{ij}$ for pairwise malware $\mathcal{G}_i$ and $\mathcal{G}_j$, firstly we set $\mathbf{A}^{ij} = \mathbf{0}$. After obtaining $\mathbf{b}$, we select the one with the largest value in it, i.e., find $t = \arg\max_t \mathbf{b}_t$, and set the pairwise matching matrix $\mathbf{A}^{ij}_{mn} = 1$, where $m = ((t-1)/|\mathcal{V}_j|+1), n = \mod((t-1), |\mathcal{V}_j|) + 1$. Then all node pairs that involve m or n will no longer be considered. Again, in the remaining pairwise node, we select the $t' = \arg\max_t \mathbf{b}_t$, where $t' \neq t, m' = ((t'-1)/|\mathcal{V}_j|+1), n' = \mod((t'-1), |\mathcal{V}_j|) + 1, m' \neq m, n \neq n'$, and set $\mathbf{A}^{ij}_{m'n'} = 1$. This process is iterated until no node can be selected from $\mathbf{b}$. If there are still unmatched nodes, they are matched with *virtual node* with null evidence values. Finally, the updated solution $\mathbf{A}$ is used for malware distance computation.

6. ENSEMBLE OF CLASSIFIERS

With learned distance metric $\mathbf{W}^q$ for each attribute type q, we next discuss how to combine results from classifiers developed for different attribute types. To this end, we use the Adaboost algorithm [6], which was proposed by Freund and Schapire, and can be used in conjunction with many other learning algorithms to further improve the classification performance. The key idea of Adaboost is that the classifiers are repeatedly improved by giving higher weights to those instances misclassified previously.

Training stage. Given the distance calculated between the attributed FCGs of two malware programs, we use the standard support vector machine (SVM) or the k-nearest neighbor classifier (kNN) for classification. For SVM, we use the Gaussian kernel, where:

$$k(\mathcal{G}_i, \mathcal{G}_j) = e^{-\gamma\frac{D^2_{i,j}}{t^2}}, \tag{18}$$

where γ is a tunable parameter, and t is the average distance of the k-nearest neighbors for each malware, which plays the role of normalization. Note that the classifier built herein deals with pairwise distances as the input for classification, rather than feature vectors as in most previous works. We further feed the similarity measures to the standard support vector machine (SVM) or the k-nearest neighbor classifier (kNN) for classification. Once we train a classifier with respect to each attribute type, we further use the Adaboost algorithm [6] to learn the confidence level associated with each classifier. Taking the SVM classifier as an example, suppose that we have obtained classification results using SVM for different attribute types on the training dataset. We then set $\mathbf{Z}_{iq} = 1$ if the SVM classifier for feature type q correctly labeled malware $\mathcal{G}_i$, otherwise $\mathbf{Z}_{iq} = 0$. Our goal is to learn the confidence level α_q for each attribute type q, such that the classification error is minimized on the training malware samples. A similar approach is adopted for the kNN classifier.

Classification stage. Having obtained the confidence level α_q associated with the classifier for each feature type q, we can use the ensemble of classifiers to classify new malware samples. Given a new malware instance, we first extract its attributed FCG $\mathcal{G}'$. Next, for each individual classifier corresponding an attribute type q, we form the evidence for this malware instance, including a set of anchor instances and their labels (i.e., the family each anchor instance belongs to), as well as the distance that the new malware sample is from each anchor instance. For instance, if the SVM classifier is used, the anchor instances are exactly those in the classifier's support vector, and for the kNN classifier, the k closest instances from the new malware sample are the anchor instances. Based on the evidence provided for each attribute type as well as the confidence level associated with each type of evidence, the ensemble of classifiers makes a decision on which family the new malware sample belongs to, where it always chooses the malware family that collects the highest total confidence weight from all the individual classifiers.

7. EXPERIMENTS

In this section, we first introduce the malware dataset used in our experiments, and then show the performance of our method.

Malware dataset. We use a malware dataset from Offensive Computing [17], which contains 526,179 unique malware variants collected in the wild. Using the VirusTotal website [24], we find that the average detection rate for 43 Anti-Virus software (such as NOD32, Symantec, McAfee, etc) is 60.5%. The malware dataset contains both packed and unpacked instances, and in our evaluation, we only use unpacked ones, and disasemble them with IDA pro [8].

Automated malware classification requires labeled malware instances for which we know their families. As reverse engineering each malware variant to obtain its family information, is a daunting task, we use majority agreement results from the five well-reputated Anti-Virus Software, McAfee, NOD32, Kaspersky, Microsoft, and Symantec. If more than three of them classify a malware into the same family, we label this malware as a variant belonging to this family. Through this way, we obtain 11 families of malware: `Bagle(Ba)`, `Bifrose(Bi)`, `Ldpinch(Ld)`, `Swizzor(Sw)`, `Zbot(Zb)`, `Koobface(Ko)`, `Lmir(Lm)`, `Rbot(Rb)`,`Sdbot(Sd)`, `Vundo(Vu)`, `Zlob(Zl)`. All these malware target the `Windows` system, and they belong to different categories, including worms, backdoor trojans, multi-component malware, etc. These malware have diverse functionalities, such as stealing user data, connection to remote IP addresses, establishment of IRC communications, etc.

Number of local functions in FCGs In Fig.(5), we show the mean and standard deviation of the number of local functions in the FCGs for each malware family. We note that the number of local functions for the `Lmir` family varies more significantly than other families. Clearly, simple statistical test using the number of

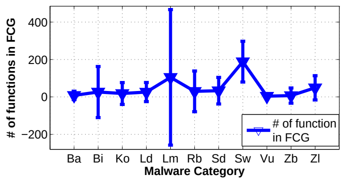


Figure 5: Mean number of functions per sample (one standard deviation).

Table 2: Different scenarios used in our experiments

Scenario	**op-n**	**mem-n**	**reg-n**	**io-n**	**flag-n**	**api-n**	**ES-ndis**
Attribute	Opcode	Memory	Register	I/O	Flag	API	Ensemble
Distance learning	No	No	No	No	No	No	No
Scenario	**op-d**	**mem-d**	**reg-d**	**io-d**	**flag-d**	**api-d**	**ES-dis**
Attribute	Opcode	Memory	Register	I/O	Flag	API	Ensemble
Distance learning	Yes	Yes	Yes	Yes	Yes	Yes	Yes

function nodes is not sufficient for identifying all malware families. Next, we show the performance of our proposed method, which exploits the structural information in these malware instances for automated malware classification.

7.1 Evaluation of our proposed method

Five-fold cross validation: In the experiments, we use 80% of the malware samples from each family to train our model, and the remaining ones are used for testing the effectiveness of our approach. This process is iterated for five times, and we report the averages as the classification performance.

Performance evaluation: The performance of a classifier can be quantified with *precision*, *recall*, and F_1. Let the number of true positives, false positives, true negatives, and false negatives be n_{tp}, n_{fp}, n_{tn} and n_{fn}, respectively, w.r.t, a classifier. The precision metric is defined to be $\frac{n_{tp}}{n_{tp}+n_{fp}}$, and the recall metric is $\frac{n_{tp}}{n_{tp}+n_{fn}}$. The F_1 measure is the harmonic mean of precision and recall, i.e., $F_1 = \frac{2n_{tp}}{2n_{tp}+n_{fp}+n_{fn}}$. An ideal classifier should have F_1 metric close to 1, implying that both precision and recall are close to 1.

Parameter settings: For the kNN classifier, we choose k between 6 and 10 in our experiments. For malware similarity computation of Eq. (18), we choose γ between 0.3 and 0.7 in one set of experiments, and t is computed using the three nearest neighbors.

7.1.1 Performance comparison

As the key components of our proposed method consists of distance metric learning and ensemble of weighted classifiers, we compare the performances of the methods in different scenarios as shown in Table 2. It is noted that our method corresponds to the **ES-dis** scenario. The distinction among these different scenarios is helpful for us to understand the origin of performance improvement.

Overall improvement: We show the F_1 measure for each family of malware in Figs. (6a, 6b) at $k = 6$ and 10, respectively, using the kNN classifier, and in Figs. (6d, 6e) at $\gamma = 0.3$ and 0.7, respectively, for the SVM classifier. In Table 3, we show the average performance improvement across all malware families. Clearly, for both classifiers, the F_1 measure is significantly improved using our method (i.e., `ES-dis`). For instance, considering both the average cases, our method improves over the best individual method by 9.3% for the SVM classifier, and by 23.2% for the kNN classifier.

Breakdown of performance improvement: In order to show how the performance improvement of our proposed method attributes to the two steps involved, distance metric learning and ensemble learning, we present in Table 4 the average F-1 measures for scenarios when distance learning is performed on individual attribute types, as well as those scenarios when the ensemble learning is used to combine individual classifiers without distance metric learning. We make two important observations from the results. First, even for the individual classifier trained for a single attribution type, distance metric learning significantly improves the classification performance, except a few cases (e.g., when kNN is used and the attribute type is opcode, memory, or register). This suggests that distance metric learning indeed helps separate malware instances belonging to different families, which eventually improves classification performance. Second, although the ensemble of classifiers does not provide significant performance improvement over the best classifier trained for a single attribute type, it does have the capability of approaching the performance of the best individual classifier trained for a single attribute. This is important, because from Figure 6 we observe that no individual classifier trained for a single attribute type is able to provide the best classification performance for different malware families on a consistent basis. Hence, ensemble learning has the advantages of finding the best-performed individual classifier, and even provides slight performance improvement over it.

Table 3: Average F-1 measure in terms of percentage across all families. For SVM, we show the results when $\gamma = 0.3$, $\gamma = 0.7$, and also the average when γ is chosen from $[0.1, 0.3, \cdots, 1.9]$; for kNN, we show the results when $k = 6$, 10, and also the average when k is chosen from [2, 4, ..., 16].

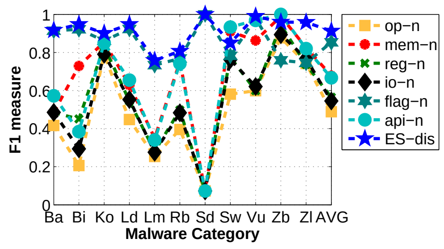
classifier	γ/k	op-n	mem-n	reg-n	io-n	flag-n	api-n	ES-dis
SVM	0.3	86.06	84.20	**86.14**	85.07	68.59	82.99	**93.88**
kNN	6	48.89	67.82	56.52	54.53	**85.33**	66.71	**91.23**
SVM	0.7	**88.61**	87.49	87.76	87.63	72.25	82.65	**98.73**
kNN	10	62.33	66.30	**87.79**	58.26	66.11	68.90	**95.31**
SVM	avg	84.54	84.51	83.57	**85.52**	68.39	84.72	**93.44**
kNN	avg	54.28	65.77	69.04	55.25	**73.52**	66.39	**90.54**

Table 4: Analysis of performance improvement. Values are shown in percentage when $\gamma = 0.7$ for SVM and when $k = 10$ for kNN.

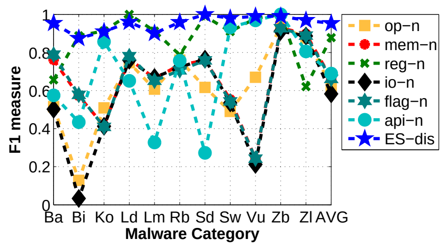
Scenario	op-n	mem-n	reg-n	io-n	flag-n	api-n	ES-ndis
SVM	**88.61**	87.49	87.76	87.63	72.25	82.65	**90.86**
kNN	62.33	66.30	**87.79**	58.26	66.11	68.90	**89.32**
Scenario	op-d	mem-d	reg-d	io-d	flag-d	api-d	ES-dis
SVM	**97.32**	94.80	96.91	97.04	97.39	95.05	**98.73**
kNN	63.24	67.05	89.15	63.05	87.03	**91.92**	**95.31**

7.1.2 Effects of parameters γ and k

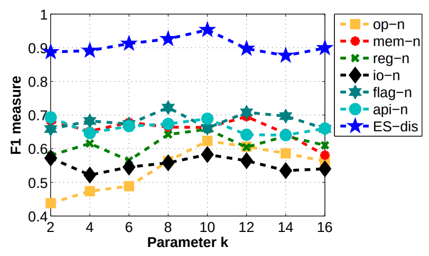
In the following, we discuss how different parameter setting of γ and k affects the performance of the SVM and kNN classifiers, respectively. In a new set of experiments, we choose the parameter γ from $[0.1, 0.3, 0.5, 0.7, \cdots, 1.5, 1.7, 1.9]$ in the gaussian kernel used for the SVM classifier, and k from $[2, 4, 6, 8, 10, 12, 14, 16]$ for the kNN classifier. The F_1 measures are shown in Fig. 6, where both distance metric learning and ensemble learning are performed in all these experiments. Clearly, irrespective of γ and k chosen for the SVM and kNN classifier, respectively, our proposed method improves the classification performance over the individual classifier trained for each feature type. Moreover, we observe that there is no apparent trend of change of classification performance when we increase parameter γ (or k) for the SVM (or kNN) classifier. Actually, for the individual classifier trained for a single attribute type, we find that for some attribute types (e.g., opcode features for kNN and flag feature for SVM), changing the parameter of the classifier leads to unstable classification performance. This may be due to the majority agreement kNN used in our approach, where for some malware samples, its k nearest neighbors may vary greatly.



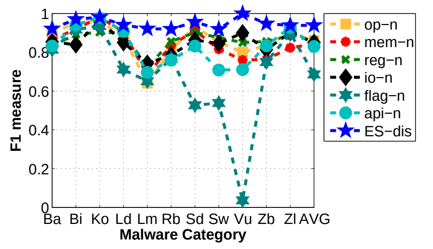
(a) (SF-ndis, ES-dis), $k = 6$, kNN



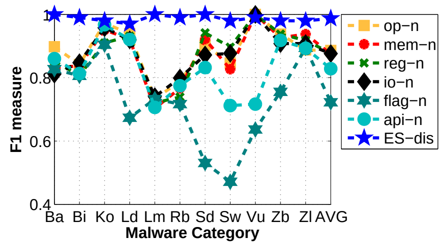
(b) (SF-ndis, ES-dis), $k = 10$, kNN



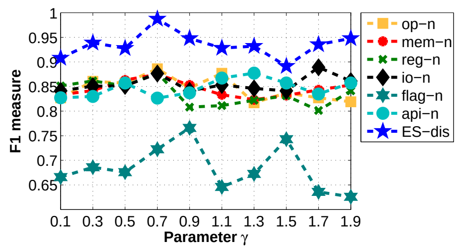
(c) Effects of k on performance of kNN



(d) (SF-ndis, ES-dis), $\gamma = 0.3$, SVM



(e) (SF-ndis, ES-dis), $\gamma = 0.7$, SVM



(f) Effects of γ on performance of SVM

Figure 6: (a-f) Performance comparison under different scenarios. `SF-ndis` means using a single feature type without distance metric learning (i.e., op-n, mem-n, etc), and `ES-dis` means using ensembles of feature types with distance metric learning. We consider both the kNN and SVM classifiers. For the former, in (a-b), we vary k between 6 and 10, and for the latter, in (d-e), we choose γ between 0.3 and 0.7. (e-f) Effects of classification parameters on kNN and SVM classifiers.

	Ba	**Bi**	**Ko**	**Ld**	**Lm**	**Rd**	**Sd**	**Sw**	**Vu**	**Zb**	**Zl**
Ba	**0.82**	0	0.04	0	0	0	0	0.14	0	0	0
Bi	0	**0.96**	0	0	0	0.02	0	0.02	0	0	0
Ko	0	0.02	**0.98**	0	0	0	0	0	0	0	0
Ld	0	0	0.04	**0.9**	0.06	0	0	0	0	0	0
Lm	0	0	0	0.1	**0.74**	0.16	0	0	0	0	0
Rd	0	0	0	0	0	**0.82**	0.18	0	0	0	0
Sd	0	0	0	0	0	0.12	**0.88**	0	0	0	0
Sw	0	0	0	0	0	0	0	**0.96**	0	0.02	0.02
Vu	0	0	0	0	0	0	0	0	**1**	0	0
Zb	0	0	0	0.02	0.04	0	0	0	0	**0.9**	0.04
Zl	0	0	0.02	0	0.06	0	0	0.02	0	0	**0.9**

Figure 7: Confusion matrix for the SVM classifier with $\gamma = 0.1$

For SVM classifier, it is known that it can be influenced a lot by different kernel parameters.

7.1.3 Cross family analysis

We show in Fig. 7 the confusion matrix, which depicts how malware instances belonging to each family are classified – or misclassified – into different families. A confusion matrix is a tabular layout in which each column corresponds to a predicted class while each row represents a real class of instances. The (i, j) element of a confusion matrix shows the percentage of instances from class i are labeled as class j. Hence, the diagonals of the confusion matrix (i.e., the (i, i) elements) provide the percentages of correctly labeled instances, whereas the off-diagonals indicate the percentages of incorrectly labeled ones.

It is known that the source code of Sdbot was published in the Internet, and the development of Rbot was influenced by it [22]. This is verified in our results, as shown in Fig. 7, where 18% of `Rbot` instances are labeled as `Sdbot` and 12% of `Sdbot`instances are labeled as `Rbot`. Moreover, for the `Lmir` family, the number of local function nodes in its attributed FCGs is highly divergent across different instances, as seen from Fig. 5, which adversely affects the computation of intra-family malware distances. This explains its poor classification performance relative to the other families.

7.1.4 Classifying samples not in known families

In practice, a newly captured executable program may not belong to the malware families which are already known to us. It can come from an unknown malware family, or even be a legitimate program. Our proposed framework can be easily extended to deal with both scenarios with little extra computation cost. In the following we show how to use the standard kNN classifier for classifying an unknown malware variant or a legimate program. As discussed above, once we learn the distance metric, for each malware family, we obtain the shortest distance between any two labeled instances in this family. Given a new sample $\mathbf{x}_t$, if it is classified as family A by our method, we further check its distance from its nearest neighbor $\mathbf{y}_t$ in this family. If the distance $d(\mathbf{x}_t, \mathbf{y}_t) > (1 + \epsilon)\theta_A$, where θ_A is the smallest distance between any two labeled samples in family A and ϵ is a tunable parameter, then we flag the new sample as one not belonging to any known malware family. The smaller parameter ϵ is, a sample that does not belong to any known family will be detected as such with a higher probability, but a sample that indeed belongs to any known family is also more likely to be misclassified.

We further perform two sets of experiments. In the first one, we choose 30 samples from the `Hupigon` family, and the second one has 15 benign executable programs. ϵ is set to be 0 and the classification accuracy (shown as percentages) is as follows:

Test samples	Hupigon	Benign
Accuracy	86.67	93.33

Clearly, our scheme can detect samples that do not belong to known families with high accuracy.

8. RELATED WORK

Since the seminal works done by Schultz *et al.* [21] and Kolter *et al.* [11], machine learning has been used in a number of efforts to automatically distinguish malware from benign executable programs (e.g., [18, 20]). In contrast to these earlier works on malware detection, our study focuses on malware classification, which aims to distinguish instances belonging to different malware families.

Even if we treat malware detection as a binary classification problem, the machine learning method proposed in this work is still unique: rather than using distance metrics that are predefined in an ad-hoc way, we learn the distance metrics in order to separate different malware families with large margin. Our experimental results tell us that distance metric learning plays an important role in improving the overall performance of automated malware classification. In [16], Nataraj *et al.* conducted a comparative assessment of malware classification using binary texture analysis and dynamic analysis, and found that binary texture analysis performs as equally effectively as dynamic analysis but is much more efficient than dynamic analysis. Although we did not compare our proposed method against the two techniques they studied, we believe that in order to improve the robustness of automated malware classification, we should consider using a combination of malware features extracted from malware programs. Hence, the structural information of malware programs considered in this work adds another layer of protection to those techniques that rely solely on binary representations of the malware programs (e.g., binary texture analysis [16]) or features extracted from dynamic execution traces.

Most existing works on malware detection and classification [18, 20] use vector features, which are amenable to traditional classification techniques such as SVM and kNN. A few other efforts (*e.g.*, [7, 14, 10, 1, 12, 13]) also considered using the structure information in malware programs. For instance, Hu *et al.* used FCGs extracted from malware programs for fast indexing, which aims to find the nearest neighbors of a new malware sample [7], and Kruegel *et al.* formulated the problem of polymorphic worm detection as coloring of control flow graphs [14]. In [25], Yan *et al.* compared the discriminative power of different types of malware features for automated malware classification. Our work differs from these efforts not only because our goal is to automatically classifying instances into their corresponding families but also our method is more generic as it learns malware distance metrics based on the structural information of labeled malware programs rather than using a predefined metric to evaluate malware similarity.

Another direction for malware research is clustering malware instances to identify groups of malware that share similar characteristics [3, 4, 9]. In contrast to malware clustering, automated malware classification trains malware classifiers from labeled data. Hence, the malware distance metric learning method proposed in this work is not suitable for malware clustering, as it requires labeled data to guide how to separate malware families with large margin. Some other contributions made in this work, such as FCG-driven feature extraction and pairwise graph matching, could be used for malware clustering as well.

9. CONCLUSIONS

In this paper, we present a generic framework that exploits the rich structural information inherent in malware programs for automated malware classification. Towards this end, we use the function call graph of a malware program to drive the process of feature extraction. We develop methods to compute the similarity of two malware programs based on their attributed function call graphs, and use an ensemble of classifiers that learn from the pairwise malware distances and classify new malware instances automatically. In the future, we plan to improve the process of obtaining labeled samples for bootstrapping automated malware classification and will also consider transductive malware classification when only limited labeled samples are available.

Acknowledgment The authors acknowledge and appreciate the support provided for this work by the Los Alamos National Laboratory Directed Research and Development Program (LDRD project 20120443ER).

10. REFERENCES

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

[2] B. Anderson, C. Storlie, and T. Lane. Improving malware classification: bridging the static/dynamic gap. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, AISec '12, pages 3–14, New York, NY, USA, 2012. ACM.

[3] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *RAID*, pages 178–197, 2007.

[4] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[5] A. P. Dempster, N. M. Laird, and D. B. Rubin. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):pp. 1–38, 1977.

[6] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT*, pages 23–37, 1995.

[7] X. Hu, T. cker Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *CCS*, 2009.

[8] `http://www.hex-rays.com/products/ida/index.shtml`.

[9] Y. Jang, D. Brumley, and S. Venkatartaman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *ACM Conference on Computer and Comunication Security*, 2011.

[10] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, 7(4):233–245, 2011.

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

[12] D. Kong, Y. Jhi, T. Gong, S. Zhu, P. Liu, and H. Xi. Sas: Semantics aware signature generation for polymorphic worm detection. In *SecureComm*, pages 1–19, 2010.

[13] D. Kong, D. Tian, P. Liu, and D. Wu. SA3: Automatic semantic aware attribution analysis of remote exploits. In *SecureComm*, 2011.

[14] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, pages 207–226. Springer-Verlag, 2005.

[15] Microsoft security intelligence report, January-June 2006.

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

[17] `http://www.offensivecomputing.net/`. Accessed in March 2012.

[18] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *ACSAC*, pages 301–310, 2008.

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

[20] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

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

[22] `http://www.honeynet.org/node/53`.

[23] 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*.

[24] `http://www.virustotal.com/`.

[25] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *Proceedings of DIMVA'13*.

[26] H. Zha, X. He, C. H. Q. Ding, M. Gu, and H. D. Simon. Spectral relaxation for k-means clustering. In *NIPS*, pages 1057–1064, 2001.