Efficient Malware Packer Identification Using Support Vector Machines with Spectrum Kernel

Tao Ban
National Institute of
Information and Communications
Technology
Koganei, Tokyo, Japan
Email: bantao@nict.go.jp

Ryoichi Isawa
National Institute of
Information and Communications
Technology
Koganei, Tokyo, Japan
Email: isawa@nict.go.jp

Shanqing Guo
Shandong University,
Jinan, China
Email: guoshanqing@sdu.edu.cn

Daisuke Inoue
National Institute of
Information and Communications
Technology
Koganei, Tokyo, Japan
Email: dai@nict.go.jp

Koji Nakao
National Institute of
Information and Communications
Technology
Koganei, Tokyo, Japan
Email: ko-nakao@nict.go.jp

***Abstract*—Packing is among the most popular obfuscation techniques to impede anti-virus scanners from successfully detecting malware. Efficient and automatic packer identification is an essential step to perform attack on ever increasing malware databases. In this paper we present a p-spectrum induced linear Support Vector Machine to implement an automated packer identification with good accuracy and scalability. The efficacy and efficiency of the method is evaluated on a dataset composed of 3228 packed files created by 25 packers with near-perfect identification results reported. This method can help to improve the scanning efficiency of anti-virus products and ease efficient back-end malware research.**

I. Introduction

Malicious software, or *malware*, plays a part in most of today's computer intrusion and security incidents. To evade detection by signature-based anti-virus (AV) scanners, an increasing percentage of malware programs distributed in the wild are packed by *packers* – software tools that transform an input executable file's appearance into another form without affecting its execution semantics [1]. The easiness to compose new variants of a well studied malware program by recursively applying a handful collection of packers has led to an exponential increase in diversity of malware programs. This, in turn, results in an exponential increase in AV signature size, seriously degrading the effectiveness of signature-based AV scanners: when an AV vendor cannot effectively unpack a packed threat, it has no choice but to create a separate signature for the threat.

A common solution to the packer problem is to extract the original code from the packed program using an appropriate *unpacker* prior to malware analysis. To this end, numerous unpackers are devised based on the reverse engineering on the packers. For popular packers such as ASPack [2], UPX [3], FSG [4], and Upack [5], there are usually multiple unpacker counterparts. Generally, unpacking is done by monitoring the execution process of the program and capturing the memory snapshot at a right timing. As an unguided unpacking attempt will expose the system to potential malware damage, the unpacking operation shall be performed in an isolated sandbox environment that supports prompt system recovery. Given the large pool of available unpackers and the hardships in manipulating their proprietary *graphical user interfaces* (GUIs), directly testing a malware database of thousands of specimens against all available unpackers will entail significant engineering and computing efforts.

To reduce the cost of unnecessary unpacking operations, *packer identification* – an intermediate step to diagnose the packer that created the given program without physically executing the program – is suggested in recent studies. Accurate and efficient packer identification can not only benefit a real-time AV scanner by reducing the computation overhead induced by the unpacking process – given the appropriate unpacking routine at hand, but provides valuable hints to back-end human analysts when the unpacking routine is unavailable, as well. Notable previous work on packer identification attributes to signature based approaches popularized by PEiD [6] and machine learning based approaches introduced in [7], [8], [9], [10].

In a recent work by Ban et al. [11], a Support Vector Machine based on Levenshtein-distance (LD) kernel (LDSVM) is introduced for packer identification. First, the most packer-relevant parts, i.e., the entry point containing sections, of packed programs with known packer information are collected to form a training dataset. Then, a radio basis function (RBF) kernel induced by LD between code segments is employed as the proximity metric to evaluate the similarity between packers. Finally, an SVM classifier built from the training dataset, with the parameters tuned by cross validation, is used to predict the packer of test malware specimens. LDSVM has presented a promising scheme to bring together the advantage of signature-based and machine learning based approaches, duly approved by its preferable performance in empirical studies. Despite of

978-0-7695-5075-6/13 $26.00 © 2013 IEEE
DOI 10.1109/ASIAJCIS.2013.18

its extraordinary generalization performance, LDSVM imposes high computing cost in training and testing. Let N be the number of programs in the training dataset, and L the length of the input code segments. The time complexity of LDSVM in training and testing is $O(L^2N^3)$ and $O(L^2N)$, respectively (See Section II-C2). While high computing cost in training will elongate the system update time at back end, high cost in testing will deter the application of LDSVM in end-user systems as well.

In this paper, to alleviate the high computing cost of LDSVM for packer identification, we propose to incorporate the popular spectrum kernel [12] with SVM. The spectrum kernel, also well known as n-gram kernel, has enjoyed great popularity due to its simplicity and efficiency in modelling processes where contiguity between items plays an important role. The advantages of adopting a spectrum kernel instead of LD kernel is summarized in the following aspects. First, it helps to save computation cost in training and testing, resulting in improved scalability and usability. The time complexity in training an SVM with spectrum of order p is $O(L\log(L) + NL)$, indicating a significant boost on the scalability of a packer identification system. On the other hand, the time complexity for predicting the packer of an previously unknown program is $O(L\log(L))$, which is a reasonable cost for real-time responding end-user clients. Second, it enables the exploration of signatures that are located apart from the starting point of the entry point containing section, and therefore could cope with packers with more complicated nature. Finally, treatment of the unique p-spectrum as independent features enables the application of advanced feature selection methods, which can not only enhance the accuracy of the system but also provide valuable hints to discover the most representative signatures of packers.

The remainder of the paper is organized as follows. In Section 2, we present related work and background information regarding packer identification. In Section 3, we present SVM with spectrum kernel function. In Section 4, we provide numerical comparison between the proposed scheme and related work. In section 5, we draw concluding remarks.

II. Related Work

Packer identification is usually performed by careful examination of packer-identifying features in a packed file. In this section, before introducing related work in packer identification, we present basic background knowledge of packers.

In [10], Sun et al. proposed a static-analysis based scheme for packer identification i.e., with the binary programs taken as the input to the system, the analysis is performed without actually executing the programs. Their study shows that the packer can be identified with good accuracy using classifiers such as k nearest neighbors (kNN), support vector machine (SVM), and naive Bayes (NB) trained from statical features extracted from the binary programs (See section II-C).

A. Background

A packer is a software program that compresses an executable file and combines the compressed data with a decompressing routine into a single executable file so that one can seamlessly execute the packed file without having to decompress it beforehand. While such a technique can be used by regular software developers for optimization purposes, many malware authors use it to obfuscate their programs in order to prevent analysis.

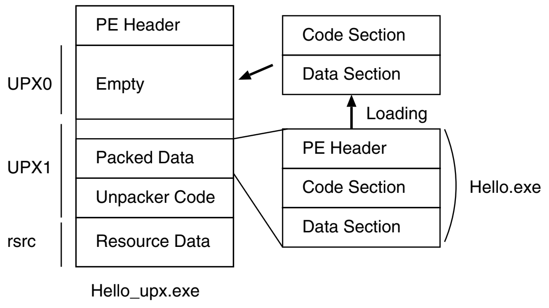


Fig. 1. The structure of a file packed by UPX.

We take UPX [3] as an example to investigate the way that a packer works. Fig. 1 shows how UPX transforms a file 'Hello.exe' into a packed file 'Hello_upx.exe' [1]. When packing an executable file, UPX merges and compresses all its three sections – the PE header, the code section, and the data section – into one data block. It then attaches a new PE header, an empty block padded with 0's, a unpacker-code block, and a resource block to the packed data. As shown in Fig. 1, the empty area, packed data together with the unpacker code, and the resource block are labeled as 'UPX0', 'UPX1', and 'rsrc', respectively. Note that since a packer author can name each section as he likes, no packer identification shall be done based on information such as section names.

The PE header of the packed file contains an *entry point* where it begins running. UPX puts the beginning address of the unpacker code into the entry point so that the unpacker code is executed at first. Then the unpacker code decompresses the original code section and data section from the packed data and overwrites the 'UPX0' section using these two sections. The unpacker code passes over to the original code after the unpacking is done.

B. Signature-based Packer Identification Tool

Since the original code of a packed file is usually compressed or encrypted, an analyst cannot perform the analysis unless it is unpacked. Packer identification when successfully done can give valuable clues on how to unpack such obfuscated specimens.

PEiD [6] is a widely used free-ware for packer identification using exact match on predefined signatures. Given a set of predefined packer signatures, e.g., one of the signatures for UPX version 2.90 is '`60 BE ?? ?? ?? ?? 8D BE ··· 11 C0 01 DB`', where each byte is presented as a bidigitate hexadecimal number and '`??`' indicates a wild card, PEiD searches over the packed file for the packer-specific signatures and outputs the name of the packer at the first hit. PEiD supports three different scanning methods, each suitable for a distinct purpose. The *normal mode* scans the specified PE file at its entry point for all designated signatures. The *deep mode* scans the file's entry point containing section. The *hardcore mode* scans the entire file for all the documented signatures.

Empirical study on the three scan modes reveals that besides its minimal computation cost, the normal mode tends to have less *false positives* than the other two modes and therefore enjoys better overall accuracy. This suggests that the entry-point containing section, especially its starting segment, carries most relevant information with regards to the packer.

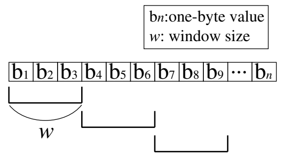
PEiD has enjoyed great popularity because of the simplicity and explicitness in its way to retrieve the packer information. However, it suffers from two open problems when applied to analyze a large scale malware collection. The first is the lack of an automated tool to extract the signatures for new packers. The high cost and clumsiness in manually extracting the signatures for new packers results in a gradual shrinkage in coverage and a degeneration in accuracy of available database. The second lies in the simple fact that PEiD performs packer identification using exact string match between its signatures and the input file. Because the result of exact signature matching usually depends heavily on the sensitivity of the signatures, definition for a new packer usually needs profound knowledge on the packer. This suggests the adoption of machine learning (ML) methods to implement an automated and tunable scheme for packer identification.

C. Machine Learning based Approach

Machine learning techniques have recently been proven to be effective in identifying packing algorithms. Early stage ML-related works focused on *packer detection* – to detect whether a program has been packed [7], [8], [9]. In practice, in addition to packer detection, further knowledge on the packer that created the given program is more helpful to the analysis.

1) Classification based on Randomness Statistics: As the extension of early stage work on packer detection, Sun et al. propose to solve the packer identification problem using an automated classification system [10]. They introduce a simple way to extract packer-discriminating features from binary programs that can serve as inputs to general classifiers such as kNN, SVM, and Naive Bayes. Feature extraction in their approach is done in three steps. In the first step, a byte-frequency histogram is created for all byte values in a given binary file and a Huffman tree [13] is built according to the histogram. This Huffman tree serves as a dictionary for the next step. In the second step, as illustrated in Fig. 2, the binary file is divided into code segments with a fixed length w. All bytes in each segment is transformed into a new sequence using the Huffman encoding and the length of this new sequence, namely, *encoding length*, is taken as the numerical feature representing this segment. In the third step, the encoding lengths for all the code segments obtained from the former step are arranged as an input vector to the classifier. It is argued in [10] that the encoding lengths of the leading N and the ending N segments carry the most discriminating packer information, where parameter N is denoted as the *prune size*, therefore each input file is represented as a numerical vector of length $2N$.

As encoding length is designed to quantify the randomness in the observed byte values within a code segment, the shape of the vector – when viewed in a time series fashion – illustrates the amount of randomness distributed in different parts of a binary file. By employing pattern classification algorithms to



b$_n$:one-byte value
w: window size
b$_1$ b$_2$ b$_3$ b$_4$ b$_5$ b$_6$ b$_7$ b$_8$ b$_9$ ··· b$_n$
w

Fig. 2. How a binary file is divided in to segments in Sun et al.'s method.

exploit the packer-discriminating information in the features, Sun et al. report promising result on a data set composed of packed binaries created by 9 packers.

2) Classification based on String Kernel: To improve the flexibility of an exact string matching metric while preserve its strength in presenting packer-specific information, Ban et al. [11] propose to use LD as the proximity metric between two files. Levenshtein Distance, also known as *edit distance* [14], is the most widely known string metric that operates between two input strings, returning a score equivalent to the number of edit operations (e.g., insertion, substitution, and deletion) needed in order to transform one input string into another. The LD distance defined upon code segments of pairwise binary programs, helps characterize the likelihood that two observed binary programs are from the same source, in our case, packed by the same packer. To make it embeddable in a kernel based algorithm such as SVM, it has to be presented in a kernel form. Ban et al. adopt the approach suggested in [15] to define an LD-kernel as

$$K(s,t) = \exp(-\gamma \cdot \delta^2(s,t)), \tag{1}$$

where γ is the width parameter of the Gaussian function that controls the nonlinearity of the kernel.

Ban et al. claim the following merits for their integrated packer-identification system: First, the LD kernel, which captures the essence of the traditional signature matching based approaches, incorporates domain knowledge on how to measure the similarity between binary programs into the learning. Second, the LD kernel introduces error tolerant mechanism to exact signature matching so that detection failure caused by minor signature violations can be significantly reduced. Finally, LDSVM takes the binary file of the program as input for the classifier, leading to substantial saving in human effort for either manual signature extraction in signature based approaches or feature distillation in feature-based approaches.

The time complexity of the LD kernel is $O(L^2)$, where L is the length of the input code segments. A state-of-the-art SVM solver requires $O(N^3)$ time, where N is the number of files in the training dataset [15]. Therefore, the overall time complexity for training an LDSVM is $O(L^2N^3)$. To compute the decision function in (7), LDs have to be evaluated between the test sample and all *support vectors*, whose number is proportional to N. Consequently, the time complexity of LDSVM for prediction is $O(L^2N)$. If the increasing scale of malware databases and pressing need to provide real-time response in end-user systems is taken into consideration, there is still much work to be done in improving the training and testing efficiency of LDSVM.

III. SVM based on Spectrum Kernel

In this section, we propose to incorporate SVM with the spectrum kernel. By the crafty implementation which will be presented, the kernel could be deemed as a linear kernel so that extra-fast linear SVM solvers can be employed in the training. In the following, introduction of SVM will be followed by the discussion on how to implement the spectrum kernel and embed it into the learning, and then by the description of adopted linear solver.

A. Support Vector Machine

The idea of a two-class Support Vector Machine (SVM) [16], [17] is described as follows. From a set of training samples $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) | \boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \{-1, +1\}, i = 1, \cdots, \ell\}$, SVM learns a norm 1 linear function, suppose its existence,

$$f(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b, \tag{2}$$

determined by a weight vector $\boldsymbol{w}$ and threshold b that realizes the maximum margin, where *margin* is defined as the distance from the hyperplane to the nearest training data point of either class. According to Vapnik's VC dimension theory, margin maximization in the training set is equivalent to maximization of the generalization error of the classifier. Two approaches could be sought in case that the training set is non-separable by a linear hyperplane. The first approach makes use of a penalty parameter C and a group of slack parameters $\xi_i (\geq 0)$ to control the trade-off between the magnitude of the margin and the error introduced by non-separable samples. The second approach suggests to first map the input vector $\boldsymbol{x}_i$ into a high (possibly infinite) dimensional feature space, $\mathcal{F}$, through a non-linear mapping function Φ, such that the improved separability between training samples from opposite classes could enforce the existence of feasible solutions of the maximum margin hyperplane in $\mathcal{F}$. With the so called *kernel trick*, the mapping Φ could be implicitly implemented by some kernel function $K(\cdot, \cdot)$, which corresponds to an inner product in the feature space, i.e.,

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \Phi(\boldsymbol{x}_i), \Phi(\boldsymbol{x}_j) \rangle, \tag{3}$$

In practice, it is typical to embrace these two approaches in the learning to gain both numerical robustness of the solution from the first approach together with adaptivity to nonlinear problems and improved separability from the second approach [18].

For a kernel-based SVM with misclassified samples being linearly penalized with a weight C, usually referred to as the *soft margin parameter*, the optimization problem can be written as:

$$\begin{cases} \min & \frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{\ell} \xi_i, \\ s.t. & y_i f(\Phi(\boldsymbol{x}_i)) \geq 1 - \xi_i, \\ & \xi_i \geq 0, \quad i = 1, \cdots, \ell, \end{cases} \tag{4}$$

The solution of the problem in (4), generally known as the *primal problem*, can be obtained with the Lagrangian theory such that $\boldsymbol{w}$ can be computed as:

$$\boldsymbol{w} = \sum_{i=1}^{\ell} \alpha_i^* y_i \Phi(\boldsymbol{x}_i), \tag{5}$$

where α_i^* is the solution of the following quadratic optimization problem, generally known as the *dual problem*:

$$\begin{cases} \max & W(\boldsymbol{\alpha}) = \sum_{i=1}^{\ell} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{\ell} \alpha_i \alpha_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j), \\ s.t. & \sum_{i=1}^{\ell} y_i \alpha_i = 0, \\ & 0 \leq \alpha_i \leq C, \quad i = 1, \cdots, \ell. \end{cases} \tag{6}$$

After training, SVM predicts the class label of a new coming test sample $\boldsymbol{x}$ based on the following decision function,

$$f(\boldsymbol{x}) = \sum_{i=1}^{\ell} \alpha_i^* y_i K(\boldsymbol{x}, \boldsymbol{x}_i) + b, \tag{7}$$

which is a kernelized version of (2). If $f(\boldsymbol{x}) > 0$, then $\boldsymbol{x}$ is assigned to the positive class, otherwise it is assigned to the negative class.

To extend SVM to solve multi-class classification problems, one may implement the "one-against-one" approach [19] or the "one-against-all" approach [16]. Take the first approach as an example, let M be the number of classes, then $M(M-1)/2$ classifiers are constructed and each one trains on data from two classes. In the prediction phase, a voting strategy is used: each binary classification is considered to be a voting where votes can be cast for all data points $\boldsymbol{x}$ – in the end a point is designated to the class with the maximum number of votes.

B. Implementing the Spectrum Kernel

The crucial observation about the dual problem (6) and the decision function (7) is that the information from the training and test samples is given by the inner products between pairwise samples. This implies the modularity of such a learning task in its dual form: on the one hand, there is no need to change the underlying algorithm to accommodate a particularly chosen kernel function; on the other hand, other types of pattern analysis could also be substituted with the chosen kernel retained. Typically, the pattern analysis algorithm component is of general purpose and the kernel function shall reflect the specific data type and domain knowledge in the data domain under discussion. Following [20], we present the spectrum kernel, which could present a proximity measure between two code segments.

Definition 1. An *alphabet* is a finite set Σ of $|\Sigma|$ symbols. A *string*

$$s = s_1 \cdots s_{|s|}, \tag{8}$$

is any finite sequence of symbols from Σ. By Σ^p we denote the set of all finite strings of length p. The string t is a substring of s if there are string u and v such that

$$s = utv. \tag{9}$$

Definition 2. The feature space $\mathcal{F}$ associated with the p-spectrum kernel is indexed by $I = \Sigma^p$, with the embedding along an axis given by

$$\phi_u^p(s) = |\{(v_1, v_2) : s = v_1 u v_2\}|, u \in \Sigma^p. \tag{10}$$

TABLE I. ALGORITHM TO REPRESENT A CODE SEGMENT IN A VECTORIAL FORM

```
function [$\boldsymbol{x}$] = vectorize($s$, $L$, $p$)
Inputs: $s$: code segment, as a pointer to char;
        $L$: Length of $s$;
        $p$: Order of the spectrum;
Outputs: $\boldsymbol{x}$: A sparse vector containing $p$-spectra in $s$;
1   $b = 8\times$(size_of_spectrum - $p$); // number of bits to shift
2   $T = \emptyset$; // a list keeping track of all spectra in $\boldsymbol{s}$
3   for $i = 0 : L - p$
4       $t$ = *typecast_char_to_spectrum($s$);
5       $t$ = shift_right(shift_left($t$, $b$), $b$);
6       $T = T \cup t$;
7       $s++$;
8   endfor
9   return histogram($T$);
```

The associated kernel is defined as

$$k_p(s_1, s_2) = \langle \phi^p(s_1), \phi^p(s_2) \rangle = \sum_{u \in \Sigma^p} \phi^p_u(s_1)\phi^p_u(s_2). \quad (11)$$

Many alternative recursions can be devised for the calculation of this kernel with different costs and levels of complexity. The cost of the implementations introduced in [20] ranges from $O(p|x_1||x_2|)$ to $O(p \max(|x_1|, |x_2|))$, which is linear in the length of the longer sequence. Due to the improved computational efficiency of the kernel, the training time of SVM will be $O(pLN^3)$ with the p-spectrum kernel. Again, here L is the length of input code segments. Because of the N^3 factor is still left in the formula, the improvement in computation efficiency as well as the scalability of the system is marginal.

When SVM is presented in the dual form as a quadratic programming (QP) problem, most QP solvers involve a slow convergence procedure during the training. Although decomposition methods such as Sequential Minimal Optimization (SMO) [21] which break down the large quadratic program method into small pieces may reduce the computation cost to some extent, the cubic factor remains. On the other hand, if the SVM could be trained in the primal form, more computationally efficient algorithms could be engaged. Notable algorithms include Joachims' cutting-plane algorithm [22], [23], the trust-region Newton method by Lin et al. [24], and the Coffin algorithm by Sonnenburg et al. [25].

Below we present our alternative approach which reduces the training time to the $O(NL)$ level. The trick to remove the N^3 factor lies in the explicit representation of the p spectrum kernel as a linear kernel in the learning. According to the definition in (11), the p-spectrum is inherently linear in nature: Let the *spectrum of order* p (p-spectrum of a code sequence x be the histogram of frequencies of the contiguous substring of length p, the *p-spectrum kernel* is the inner product of their p-spectra. Therefore, the remaining problem is how to represent the p-spectrum explicitly in a vector form. Despite of the difficulty to numerate all the possible p-spectra in Σ, the number of p-spectra presented in a code segment with length L is $L - p + 1$. When presented in a parse form (using spectrum-frequency tuples) following the algorithm in Table 1, these $L - p + 1$ spectra could be manipulated as a normal vector, ready to be applied to any solver that takes advantage of the vectorial form of the input.

The **vectorize** algorithm in Table I takes in three parameters: the code segment s as a string, the length L of s (in byte), and the order of the spectrum kernel, p. It returns a sparse vectorial representation of the code segment in $\boldsymbol{x}$, which serves as input to an linear SVM solver. The conversion is iteratively done for consequential bytes as follows. First, a casting from *pointer to byte* to *pointer to spectrum* is done using a data-type casting function (line 4). Here, the spectrum data-type could be an unsigned integer type with a sufficient capacity, e.g., commonly used 64-bit unsigned integer data type suffices for spectrum computation with p up to 8. Then it performs two times the bit-wise shift (line 5) to remove the unnecessary concatenating code in the rear. All the converted spectra from s are kept tracked in list T in the iteration from line 3 to line 7. Then by the histogram function on line 9, the spectra are sorted in ascending order and represented in a sparse vectorial form as spectrum-frequency pairs.

C. Classification Methods

The above p-spectrum kernel defined upon two code segments corresponds to a inner produced in the feature space $\mathcal{F}$. Then, the Euclidean distance between two code segments s_1 and s_2 in $\mathcal{F}$ could be computed as

$$d(s_1, s_2) = (K(s_1, s_1) + K(s_2, s_2) - \cdot K(s_1, s_2))^{\frac{1}{2}}. \quad (12)$$

With this distance metric, many classification algorithms could be applied for unpacker identification. In our proposed scheme, the linear SVM solver proposed in [24] is employed to build the model from the training data. To be brief, the following L2-SVM is solved,

$$\min_{\boldsymbol{w}} g(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{\ell}(\max(0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i)^2, \quad (13)$$

by the trust regioin Newton method in troduced in [24] and implemented in the LibLinear toolbox [26].

IV. EXPERIMENTS

We evaluate the classification performance of the proposed p-spectrum SVM (pSSVM), LD distance SVM (LDSVM), LD based kNN (LDKNN), kNN and SVM classifier based on feature extraction by Sun et al. (SUNKNN and SUNSVM, respectively). The experiments are conducted on a PC with Xeon 2.66GHz CPU, 4GB memory using C++ language.

A. Dataset

We perform packer identification on a dataset of 3228 Portable Executables (PE) files collected as follows. First, a collection of executable PE files are acquired from a clean installation of Windows XP Professional. These files are checked against the entropy based packer detector introduced in [27]. 219 files from the collection which are classified as *unpacked* are selected as the base files. These base files are given to 25 packers to generate the packed files, from which non-executable files are discarded. The executable packed file and base files are taken as the training set where the labels are determined the associated packers. See Table IV for the packer names and the number of files in each class.

TABLE II. NOTATION.

PARA.	CLASSIFIER	PHYSICAL MEANING	GRID VALUES
p	pSSVM	Order of the spectrum	$\{2, 3, \cdots, 8\}$
K	SUNKNN, LDKNN	Number of nearest neighbors for kNN	$\{1, 3, \cdots, 11\}$
W	SUNKNN, SUNSVM	Code segment length for Sun et al.'s feature extraction	$\{8, 16, 32, 64\}$
N	SUNKNN, SUNSVM	Prune size for Sun et al.'s feature extraction	$\{5, 10, \cdots, 125\}$
C	pSSVM, SUNSVM, LDSVM	Panelty parameter	$\{2^0, 2^1, \cdots, 2^14\}$
L	pSSVM, LDKNN, LDSVM	Input code length starting from entry point	$\{40, 80, \cdots, 280, 320\}$
γ_1	SUNSVM	Width parameter for RBF kernel	$\{10^{-3}, 10^{-2}, 10^{-1}, 1\}$
γ_2	LDSVM	Width parameter for LD kernel	$\{2^{-10}, 2^{-9}, \cdots, 2^0\}$

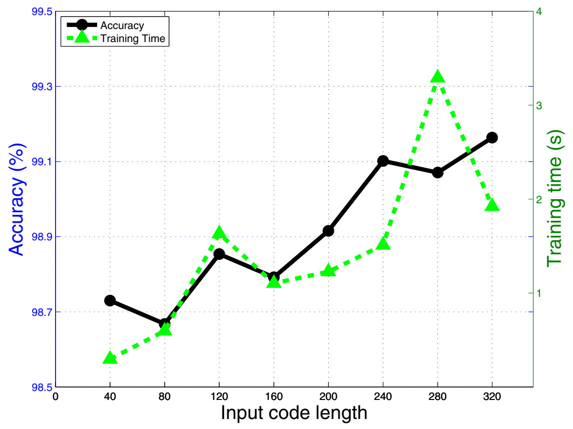


Fig. 3. Tuning parameter L using 10-fold cross validation. L is selected from $\{40, 80, 120, 160, 200, 240, 280, 320\}$.

B. Parameter Tuning

Since performance of the algorithms usually depends heavily on the parameters, we use 10-fold cross validation to tune the parameters of the compared algorithms. To do so, the training set is first randomly partitioned into 10 disjoint folds. Then, in the ith of the 10 cross-validation iterations, a classifier is trained using the complementary set of the ith fold, and tested against the ith fold. This procedure is done for each of the parameter combinations and the best cross validation result and the corresponding parameter setting are reported. As it is not possible to go over all the possible parameter combinations, in the experiment, we search over the parameter settings that falls on a sparse grid. Table II lists all the involved parameters in the experiments and all the grid values that we have examined.

As an example of the parameter tuning process, Fig. 3 shows how the length parameter L is determined for pSSVM. The blue dashed line shows the overall prediction accuracy averaged from the top ten (C, γ) settings for each L parameter. And the green solid line shows the averaged training time for the building the model using the same setting. It is easy to spot the increment in accuracy as L increase from 40 to 320. Meanwhile, the training time keeps increasing as L increases to 280, and decreases to some extent for $L = 320$. These results suggest that 320 is the most appropriate length value: it yields the best generalization performance on a wide range of (C, γ) combinations with a reasonable computing cost: training performed on 3,228 samples with 26 classes takes about 2 seconds.

To get the results reported in Table III, all parameters, including common parameters such as the length of the code segment, L, in LDKNN and LDSVM, are independently tuned for each algorithm. The parameters settings that yield the best cross validation performance are listed in the right-most column in the table.

C. Evaluation Metrics

To access the generalization performance of the aforementioned classifiers, we adopt conventionally used metrics such as *accuracy, precision, recall*, and *false positive rate*. These metrics are defined based on the following intermediate measures.

- True positive (TP): the number of predicted positive records correctly classified;
- False positive (FP): the number of predicted positive records classified wrong;
- True negative (TN): the number of predicted negative records correctly classified;
- False negative (FN): the number of predicted negative records classified wrong.

Then, the accuracy is the percentage of test records that are correctly classified, i.e.,

$$Accuracy = \frac{TP + TN}{n}; \tag{14}$$

precision is the probability that predicted positives that are correct classified, i.e.,

$$Precision = \frac{TP}{TP + FP}; \tag{15}$$

recall is the probability that the record in the class is correctly classified, i.e.,

$$Recall = \frac{TP}{TP + FN}; \tag{16}$$

and false positive rate is the probability that a negative result is falsely classified to the positive class, i.e.,

$$Recall = \frac{FP}{FP + TN}. \tag{17}$$

D. Numerical Results

In Table III, we compare the overall classification accuracy using the best results returned by 10-fold cross validation. On the dataset, SUNKNN gives an overall accuracy of 91.42% which is considerably lower than the 99.4% accuracy reported in [10] on a dataset created by 9 packers. This degeneration may be caused by the increased number of packers in creating the dataset and newer types of packers that support obfuscation

TABLE III. CROSS VALIDATION RESULTS BASED ON PARAMETER TUNING.

CLASSIFIER	ACCURACY(%)	Train. Time(s)	PARAMETERS
SUNKNN	91.42	–	$K = 1$, $W = 32$, $N = 200$
SUNSVM	93.93	30	$C = 128$, $\gamma = 0.01$, $W = 32$, $N = 180$
LDKNN	98.54	–	$K = 1$, $W = 32$, $L = 200$
LDSVM	99.35	2136.7	$C = 1024$, $\gamma = 2^{-8}$, $L = 200$
pSSVM	99.19	1.9	$C = 512$, $p = 3$, $L = 320$
PEiD	69.81	–	normal scan
PEiD	63.50	–	deep scan
PEiD	63.50	–	hardcore scan

techniques. In addition, our dataset contains an *unpacked* class, which is most challenging because it contains multi-source files that might not be represented by a handful number of signatures. On the same feature set, SUNSVM outperforms SUNKNN by a margin of 2%. This is commonly recognized in ML field as of SVM's extraordinary generalization ability.

When LD is applied as the similarity metric, kNN yields an overall accuracy of 98.64%. Due to the simple decision rule for kNN, we can attribute the improvement to LD for being able to capture the most essential discriminating information for different packers. By virtue of the good generalization ability of SVM, it is not surprising that the accuracy is further improved to 99.35% using LDSVM.

When the p-spectrum kernel is applied to the learning, we got an overall accuracy of 99.19%, which is comparable with LDSVM's generalization performance. It achieves an 7.5% percent improvement in overall accuracy as compared to SUNKNN, over performing the rest of evaluated methods. This indicates that the discriminant information captured by the p spectrum kernel is no less than the LD distance. While it retains the accuracy in the same grade of LDSVM, the computational cost is substantially reduced to the second level, speeding up the training of LDSVM by three orders of magnitude.

As a reference, the results of PEiD using all of its three scan modes are also reported in Table III. Because of missing signatures of the involved packers, PEiD fails to give comparable results to ML based methods. It worth noting that deep scan and hardcore scan of PEiD that scan much more file contents do not yield in better accuracy than normal scan because of increased false positives.

Table IV reports class specific results for PEiD (normal scan), SUNSVM, LDSVM, pSSVM. Similar as in the above comparison, pSSVM and LDSVM outperforms the other two algorithms in most of the cases. As mentioned before, the 'unpacked' class (the 8th class in the table) is one of the most difficult classes because of the variety in signatures from different compilers. All of pSSVM, SUNSVM, and LDSVM fail to give perfect result on this class. PEiD performs better than the other two methods because of a good coverage of signatures of commonly used compilers. Take the 'nPackv1.1.300' class (the 10th class in the table) as another example. Here, PEiD's signature seems to be too rigorous so that many files created by this packer are erroneously assigned to other classes resulting in an 11.2% recall rate. pSSVM and LDSVM successfully capture the characteristic of this class so that no error is made on this class. These class specific results further prove the capability of pSSVM to automatically learn the most essential packer-relevant information from the binary code sequences.

V. CONCLUSION

In this paper we presented a new approach to identify the packing technique that is used to pack a given software program. Our method makes use of p-spectrum kernel to capture the discriminating packer information at the starting section of a binary program's entry point so that high accuracy and efficiency is secured. Experiments show that the proposed approach has a comparable generalization performance with the LDSVM, outperforming PEiD and previous approaches with a large margin. It shows an extraordinary performance in terms of the training and testing and therefore lead to acceptable scalability in practical environments. As future study, we will explore the way to combine multiple features to further improve the accuracy, and apply the proposed approach to facility automated malware analysis.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 24700084.

REFERENCES

[1] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 98–115.

[2] A. software, "http://www.aspack.com/," 2007.

[3] M. F. Oberhumer, L. Molnr, and J. F. Reiser, "UPX: Ultimate Packer for eXecutables," 2007.

[4] bart, "Fsg: [f]ast [s]mall [g]ood exe packer," 2005.

[5] Dwing, "Winupack 0.39final," 2006.

[6] PEiD, most recent release (PEiD 0.95) could be downloaded from http://www.softpedia.com/.

[7] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security and Privacy*, vol. 5, pp. 40–45, March 2007.

[8] R. Perdisci, A. Lanzi, and W. Lee, "Classification of packed executables for accurate computer virus detection," *Pattern Recogn. Lett.*, vol. 29, pp. 1941–1946, October 2008.

[9] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *J. Mach. Learn. Res.*, vol. 7, pp. 2721–2744, December 2006.

[10] L. Sun, S. Versteeg, S. Boztaş, and T. Yann, "Pattern recognition techniques for the classification of malware packers," in *Proceedings of the 15th Australasian Conference on Information Security and Privacy*, ser. ACISP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 370–390.

[11] I. R. G. S. I. D. Ban, Tao and K. NAKAO, "Application of string kernel based support vector machine for malware packer identification," in *Proceedings of the 2013 International Joint Conference on Neural Networks*, ser. IJCNN'13. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

TABLE IV. CLASS SPECIFIC RESULTS.

		PEiD			SUNSVM			LDSVM			pSSVM		
Classifier	#	P(%)	R(%)	FPR(%)	P(%)	R(%)	FPR(%)	P(%)	R(%)	FPR(%)	P(%)	R(%)	FPR(%)
Armadillov4.00.0053	203	100.0	99.5	0.0	91.9	95.6	0.6	98.1	100.0	0.1	99.0	100.0	0.1
MoleboxPro2.6.4	203	100.0	99.5	0.0	95.3	100.0	0.3	100.0	100.0	0.0	100.0	100.0	0.0
Themidav1.8.5.5	195	100.0	99.5	0.0	90.0	92.3	0.7	100.0	100.0	0.0	100.0	100.0	0.0
PESpinv1.33	194	100.0	99.5	0.0	93.0	89.7	0.4	100.0	100.0	0.0	100.0	100.0	0.0
obsidium1.3.5.4	191	NaN	0.0	0.0	99.5	99.5	0.0	100.0	100.0	0.0	100.0	100.0	0.0
asprotect2.1	188	100.0	99.5	0.0	96.8	97.9	0.2	100.0	100.0	0.0	98.4	100.0	0.1
yodaprotector1.02	185	100.0	99.5	0.0	98.9	99.5	0.1	100.0	100.0	0.0	100.0	100.0	0.0
Unpacked	170	98.1	90.0	0.1	83.5	86.5	0.9	95.9	97.1	0.2	95.8	92.9	0.2
obsidium1.4.5	165	NaN	0.0	0.0	100.0	99.4	0.0	100.0	100.0	0.0	100.0	100.0	0.0
nPackv1.1.300	143	100.0	11.2	0.0	99.3	100.0	0.0	100.0	100.0	0.0	100.0	100.0	0.0
eXPressor1.5.0.1	141	NaN	0.0	0.0	95.1	96.5	0.2	100.0	100.0	0.0	100.0	100.0	0.0
ASPack2.12	139	100.0	99.3	0.0	86.0	84.2	0.6	100.0	100.0	0.0	100.0	100.0	0.0
PECompactv2.64	127	100.0	99.2	0.0	96.2	98.4	0.2	100.0	100.0	0.0	100.0	100.0	0.0
NsPackv3.7	122	100.0	99.2	0.0	97.5	94.3	0.1	100.0	100.0	0.0	100.0	100.0	0.0
RLPack1.20	116	NaN	0.0	0.0	98.3	99.1	0.1	100.0	100.0	0.0	99.1	100.0	0.0
UPX3.08	111	97.3	99.1	0.1	98.2	97.3	0.1	99.1	99.1	0.0	97.3	98.2	0.1
ASPack2.11	99	100.0	99.0	0.0	79.6	78.8	0.6	100.0	100.0	0.0	100.0	98.0	0.0
Mew11SEv1.2	99	100.0	98.0	0.0	99.0	99.0	0.0	100.0	100.0	0.0	99.0	100.0	0.0
acpr_pro_1.32	88	98.7	87.5	0.0	93.0	90.9	0.2	83.3	96.6	0.5	95.6	97.7	0.1
WWPack32.1.20	75	100.0	100.0	0.0	98.6	94.7	0.0	100.0	100.0	0.0	100.0	100.0	0.0
AntiCrack1.32Pro	64	NaN	0.0	0.0	87.3	85.9	0.3	96.0	75.0	0.1	98.4	95.3	0.0
PETITE2.2	63	100.0	96.8	0.0	90.2	87.3	0.2	100.0	96.8	0.0	96.8	96.8	0.1
exe32pack1.4.2	62	100.0	98.4	0.0	86.4	82.3	0.3	100.0	96.8	0.0	100.0	96.8	0.0
tElockv0.98	55	100.0	100.0	0.0	94.1	87.3	0.1	100.0	100.0	0.0	100.0	100.0	0.0
PKLITE32	20	100.0	90.0	0.0	61.5	40.0	0.2	100.0	90.0	0.0	85.7	90.0	0.1
Upack0.39	10	100.0	90.0	0.0	90.0	90.0	0.0	100.0	100.0	0.0	100.0	100.0	0.0

(#): number of samples, (P): precision, (R):recall, (FPR): false positive rate.

[12] E. Vegas, F. Reverter, J. M. Oller, and J. M. Elías, "A comparison of spectrum kernel machines for protein subnuclear localization," in *Proceedings of the 5th Iberian Conference on Pattern Recognition and Image Analysis*, ser. IbPRIA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 734–741.

[13] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[14] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[15] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, May 2011.

[16] V. N. Vapnik, *Statistical Learning Theory*, 1st ed. Wiley, Sep. 1998.

[17] B. Scholkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001.

[18] T. M. Cover, "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition," *Electronic Computers, IEEE Transactions on*, vol. EC-14, no. 3, pp. 326–334, 1965.

[19] S. Knerr, L. Personnaz, and G. Dreyfus, "Single-layer Learning Revisited: A Stepwise Procedure for Building and Training a Neural Network," in *Neurocomputing: Algorithms, Architectures and Applications*, J. Fogelman, Ed. Springer-Verlag, 1990.

[20] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. New York, NY, USA: Cambridge University Press, 2004.

[21] J. C. Platt, "Advances in kernel methods," B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds. Cambridge, MA, USA: MIT Press, 1999, ch. Fast Training of Support Vector Machines Using Sequential Minimal Optimization, pp. 185–208.

[22] T. Joachims, "Advances in kernel methods," B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds. Cambridge, MA, USA: MIT Press, 1999, ch. Making Large-scale Support Vector Machine Learning Practical, pp. 169–184.

[23] ——, "Training linear svms in linear time," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 217–226.

[24] C.-J. Lin, R. C. Weng, and S. S. Keerthi, "Trust region newton methods for large-scale logistic regression," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 561–568.

[25] S. Sonnenburg and V. Franc, "Coffin : A computational framework for linear svms," in *Proc. ICML 2010*, 2010.

[26] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, Jun. 2008.

[27] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 40–45, Mar. 2007.