Malware Detection based on Graph Classification*

Khanh-Huu-The Dam[1] and Tayssir Touili[2]
[1]*University Paris Diderot and LIPN, Villetaneuse, France*
[2]*LIPN, CNRS and University Paris 13, Villetaneuse, France*

Keywords: Machine Learning, Graph Kernel, Malware Detection, Static Analysis.

Abstract: Malware detection is nowadays a big challenge. The existing techniques for malware detection require a huge effort of engineering to manually extract the malicious behaviors. To avoid this tedious task of manually discovering malicious behaviors, we propose in this paper to apply learning for malware detection. Given a set of malwares and a set of benign programs, we show how learning techniques can be applied in order to detect malware. For that, we use abstract API graphs to represent programs. Abstract API graphs are graphs whose nodes are API functions and whose edges represent the order of execution of the different calls to the API functions (i.e., functions supported by the operating system). To learn malware, we apply well-known learning techniques based on Random Walk Graph Kernel (combined with Support Vector Machines). We can achieve a high detection rate with only few false alarms (98.93% for detection rate with 1.24% of false alarms). Moreover, we show that our techniques are able to detect several malwares that could not be detected by well-known and widely used antiviruses such as Avira, Kaspersky, Avast, Qihoo-360, McAfee, AVG, BitDefender, ESET-NOD32, F-Secure, Symantec or Panda.

1 INTRODUCTION

The number of malwares is significantly increasing. In 2014, there were more than 317 million new pieces of malwares[1] compared to 286 millions in 2010. It is estimated that there are nearly a million of new malwares released every day. Thus, malware detection is a big challenge.

The well-known technique to detect malware is signature matching. It consists on searching for patterns in the form of binary sequences (called signatures) in the program. Signatures are manually introduced in a database by experts. If a program contains a signature in the database, it is declared as a virus. If not, it is declared as benign. It is very easy for virus writers to get around these signature matching techniques. Indeed, obfuscation techniques can change the structure of a malware so that it will not have a known signature anymore while keeping its same behavior.

Another technique to detect malware is called dynamic analysis. It consists in running a malware in an emulated environment and recording its behaviors in real time. However, as the execution time is limited, it is hard to trigger the malicious behaviors, since these may be hidden behind user interaction or require delays.

To sidestep these limitations, static analysis techniques that allow to analyse the behavior (not the syntax) of the program without executing it were applied for malware detection (Bergeron et al., 1999; Christodorescu and Jha, 2003; Kinder et al., 2010; Song and Touili, 2013a). However, in these works, the malicious behaviors are discovered after a manual study of the assembly code of the malwares. That task needs an enormous engineering effort and takes an enormous amount of time. This is the reason why only 7 malicious behaviors were considered in (Song and Touili, 2013b), whereas there are much more malicious behaviors that should be considered. Thus, one needs techniques that prevent us from performing this enormous amount of engineering effort of reading assembly codes to discover malicious behaviors.

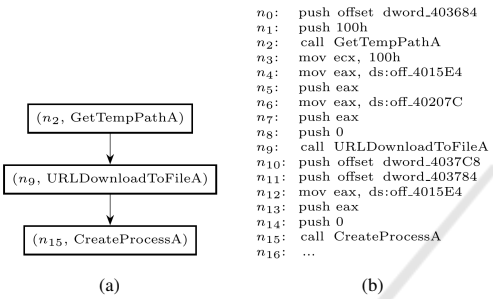
To solve this problem, we apply in this work machine learning techniques for malware detection. Given a set of malwares and a set of benign programs, we use machine learning techniques to teach computers to *automatically* learn malicious behaviors. To do this, we need an abstract representation of programs (malicious behaviors) that we have to learn. Following (Fredrikson et al., 2010; Babić et al., 2011; Macedo and Touili, 2013), we use API

*This work was partially funded by the FUI project AIC 2.0.

[1]2015 Internet Security Threat Report, Volume 20, Symantec

Dam, K-H-T. and Touili, T.
Malware Detection based on Graph Classification.
DOI: 10.5220/0006209504550463
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 455-463
ISBN: 978-989-758-209-7

function calls to specify malicious behaviors. Indeed, API (which stands for Application Programming Interface) is a collection of functions supported by the operating system that allow users to interact with the system. These API functions are mediators between programs and their running environment (user data, network access...) that are mostly used to access or modify the system by malware authors. According to a statistic study[2], over 5TB of different samples of malwares, there are 527,992 samples that did import at least one API, compared to 21,043 samples with no import. Thus, API functions and their usages in the program are crucial to specify malicious behaviors. Let us consider a typical malicious behavior.



(a) (b)

Figure 1: The assembly code fragment of a trojan downloader (b) and the API call graph (a).

Figure 1(b) is a fragment of the assembly code of a trojan downloader. First, the function `GetTempPathA` is called. This allows the program to get the location of the temporary directory in Windows OS. Then, the function `URLDownloadToFileA` is called to download a file to this directory. Finally, this file is executed by calling the function `CreateProcessA`. This is a typical behavior of a trojan downloader. In order to represent this behavior we use an API call graph, which is a graph whose vertices are pairs (n, f) consisting of an API function f and a control point n, and whose edges $((n, f),(n', f'))$ express that there is a call to the API function f at the control point n, followed by a call to the API function f' at the control point n', such that between the calls f and f', there is no other call to another API function. Figure 1(a) represents the API call graph of the behavior of the trojan downloader. The edge $((n_2,$ `GetTempPathA`$),(n_9,$ `URLDownloadToFileA`$))$ expresses that at the control point n_2 there is a call to the function `GetTempPathA` followed by the call to the function `URLDownloadToFileA` at the control point n_9. Since the size of such graphs is huge in the case of malwares, we apply an abstraction to reduce the size of these graphs by merging vertices corresponding to the same API function into one vertex associated with the function name. Such graphs are called abstract API graphs.

Using this representation, we apply machine learning techniques on graphs to learn malicious behaviors, and detect malwares. Support Vector Machine (SVM) is one of the most successful techniques in machine learning. It has been applied to several fields in pattern recognition including text analysis and bioinformatics. In this work, we apply Support Vector Machine based learning techniques for malware detection. The choice of Support Vector Machine is motivated by the fact that they are very suitable for nonvectorial data (graphs in our setting), whereas the other well-known learning techniques like artificial neural network, k-nearest neighbor, decision trees, etc. can only be applied to vectorial data. This SVM method is highly dependent on the choice of kernels. A kernel is a function which returns similarity between data. Standard kernels (including linear, polynomial, etc) handle vectorial data. However, for nonvectorial data such as graphs, these kernels become non suitable. That is the reason why we need to use specific kernels for graphs. In this work, we use a variant of the random walk graph kernel that measures graph similarity as the number of common paths of increasing lengths.

The main contribution of this paper is the application of graph kernel based learning techniques for malware detection in a completely static way (no dynamic analysis). As far as we know, this is the first time that these techniques are applied for malware detection in a static manner. We implemented our technique in a tool and tested it on a dataset of 6291 malwares, that are collected from Vx Heavens[3], and obtained encouraging results. Our tool can achieve a high detection rate with only few false alarms (98.93% for detection rate with 1.24% of false alarms).

Moreover, we show that our techniques are able to detect several malwares that could not be detected by well-known and widely used antiviruses such as Avira, Kaspersky, Avast, Qihoo-360, McAfee, AVG, BitDefender, ESET-NOD32, F-Secure, Symantec or Panda.

In this paper, we introduce our graph model in Section 3. In Section 4, we discuss Support Vector Machine techniques and the application of graph kernels to our graphs in order to detect malwares. Experiments are given in Section 5.

[2] http://www.bnxnet.com/

[3] http://vxheavens.org

2 RELATED WORK

Machine learning techniques were applied for malware classification in (Schultz et al., 2001; Kolter and Maloof, 2004; Gavrilut et al., 2009; Tahan et al., 2012; Khammas et al., 2015). However, all these works use either a vector of bits (Schultz et al., 2001; Gavrilut et al., 2009) or n-grams (Kolter and Maloof, 2004; Tahan et al., 2012; Khammas et al., 2015) to represent a program. Such vector models allow to record some chosen information from the program, they do not represent the program's behaviors. Thus they can easily be fooled by standard obfuscation techniques, whereas our API graph representation is more precise and represents the API call behavior of programs and can thus resist to several obfuscation techniques.
(Ravi and Manoharan, 2012) use sequences of API function calls to represent programs and learn malicious behaviors. Each program is represented by a sequence of API functions which are captured while executing the program. (Rieck et al., 2008) uses as model a string that records the number of occurences of every function in the program's runs. Our model is more precise and more robust than these two representations as it allows to take into account several API function sequences in the program while keeping the order of their execution. Moreover, (Ravi and Manoharan, 2012) and (Rieck et al., 2008) use dynamic analysis to extract a program's representation. As said above, our API graph extraction is done in a static way.

(Christodorescu et al., 2007; Kinable and Kostakis, 2011; Fredrikson et al., 2010; Macedo and Touili, 2013; Elhadi et al., 2015) represent programs using graphs similar to our API call graphs. (Christodorescu et al., 2007; Fredrikson et al., 2010; Macedo and Touili, 2013) use graph mining algorithms to compute the subgraphs that belong to malwares and not to benign programs and they assume that these correspond to malicious behaviors. We do not make such assumption as two malwares may not have any common subgraphs. Moreover, (Christodorescu et al., 2007; Fredrikson et al., 2010) use dynamic analysis to compute the graphs, whereas our graph extraction is made statically. (Kinable and Kostakis, 2011) uses clustering techniques. This approach depends highly on the number of clusters that has to be provided. The performance degrades if the number of clusters is not optimal. (Elhadi et al., 2015) uses graph similarity based on comparison of the longest common subsequences. Our graph kernels are more robust since, to compare graphs, we take into account all paths existing in the graph.
(Nikolopoulos and Polenakis, 2016) use graphs similar to our API graphs where each node corresponds to a group of API function calls. Our graphs are more precise since we do not group API functions together. Moreover, (Nikolopoulos and Polenakis, 2016) uses dynamic analysis to extract graphs, whereas our techniques are static. Furthermore, they define their own similarity metric to classify malwares whereas we use the well-known SVM method for malware classification.
(Kong and Yan, 2013; Xu et al., 2013) use graphs where nodes are functions of the program (either API functions or any other function of the program). Such representations can easily be fooled by obfuscation techniques such as function renaming. Moreover, these works do not use graph kernel based SVM to classify graphs.

Graph kernel based SVM for malware detection is used in (Anderson et al., 2011; Wagner et al., 2009). (Wagner et al., 2009) uses graphs to represent the system's behaviors (system commands, process IDs...) not the program's behaviors as we do. This approach can only be done by dynamic analysis. Moreover, (Wagner et al., 2009) uses a kind of random walk graph kernel based SVM to learn malicious behaviors. Our random walk graph kernel is more precise for graph comparison since our kernel takes into account path lengths in graphs in a more precise way. As for (Anderson et al., 2011), they use graphs to represent the order of execution of the different instructions of the programs (not only API function calls). Our API graph representation is more robust. Indeed, considering all the instructions in the program makes the representation very sensitive to basic obfuscation techniques. Moreover, (Anderson et al., 2011) uses graph kernel based SVM to learn malicious behaviors. They use the Gaussian and spectral kernels which allow them to compare the structure of graphs. Our random walk graph kernel compares the paths of the graph instead. This allows us to compare the behaviors of the programs where a behavior is a sequence of API functions.

3 BINARY CODE MODELING

Malwares are usually executables, i.e., binary codes. Thus, we show in this section how to extract an API call graph from a binary code. Given a binary code, we apply the disassembly tools IDA Pro (Eagle, 2011), Jakstab (Kinder and Veith, 2008) and BePum (Nguyen et al., 2013) to extract a control flow graph (CFG) (a standard representation of programs in the program analysis community). Then, we use this CFG to construct an API call graph. Since mal-

wares contain a huge number of instructions in their codes, the obtained API call graphs are huge (more than 854 vertices in our dataset). Thus, the learning technique we applied took a lot of time. To be more efficient, we introduce an abstraction of the API call graph, called abstract API graph, that consists in merging the vertices of the API call graph that correspond to the same API function. In this section, we first recall the definition of a control flow graph, then we define API call graphs and abstract API graphs, and show how to compute them from the CFG of the program.

3.1 Control Flow Graph

A Control Flow Graph (CFG) is a tuple $G = (N, I, E)$, where N is a finite set of vertices, I is a finite set of assembly instructions in a program, and $E : N \times I \times N$ is a finite set of edges. Each vertex corresponds to a control point of the program. Each edge connects two control points in the program and is associated with an assembly instruction. An edge (n_1, i, n_2) in E expresses that in the program, the control point n_1 is followed by the control point n_2 and is associated with the instruction i.

3.2 API Call Graph

Let $\mathcal{A}$ be the set of all API functions that are called in the program. An API call graph is a directed graph $G_{api} = (V_{api}, E_{api})$, where $V_{api} : N \times \mathcal{A}$ is a finite set of vertices and $E_{api} : (N \times \mathcal{A}) \times (N \times \mathcal{A})$ is a finite set of edges. We define the labeling function $\ell : V_{api} \rightarrow \mathcal{A}$ such that $\ell((n, f)) = f$ (the label of a vertex is its corresponding API function). A vertex (n, f) means that at a control point n, a call to the API function f is made. An edge $((n_1, f_1), (n_2, f_2))$ in E means that the API function f_2 called at the control point n_2 is executed after the API function f_1 called at the control point n_1. Moreover, between the control points n_1 and n_2, there is no call to another API function.

3.3 Abstract API Graph

As the size of the previous graph is quite huge in the case of malwares, we apply an abstraction to reduce the size of the API call graphs by merging vertices corresponding to the same API function in one vertex associated with the function name, i.e., the vertices (n_1, f), (n_2, f), ..., (n_k, f) are merged in a single vertex labeled by the API function f. By doing that, the size of graphs in our dataset is reduced by about a quarter while the accuracy is not changed so much. Thus, all our experiments are made on abstract API graphs (not on API call graphs).

Given an API call graph $G_{api} = (V_{api}, E_{api})$, an abstract API graph is a directed graph $G_{aapi} = (V_{aapi}, E_{aapi})$, where $V_{aapi} \subseteq \mathcal{A}$ is a set of vertices, and E_{aapi} is a set of edges. Each vertex is labeled by an API function. There is an edge $(f_1, f_2) \in E_{aapi}$ if there exist control points n_1 and n_2, such that $((n_1, f_1), (n_2, f_2))$ is in G_{api}. We define the labeling function $\ell : V_{aapi} \rightarrow \mathcal{A}$ such that $\ell(v) = v$ for every $v \in V_{aapi}$ (the label of a vertex is its corresponding API function).

4 LEARNING MALICIOUS BEHAVIORS

In order to detect malicious behaviors, we cast the problem of malware detection as graph classification. The goal is to check whether a given unseen data[4] belongs to the positive (malign) or the negative (benign) class. For that purpose, we build a classifier, that decides about this class membership using a labeled training set. The latter includes positive as well as negative examples.

In what follows, we discuss the application of kernel-based support vector machines (SVMs) in malware detection. The choice of SVMs is motivated by their well established generalization ability in many pattern classification problems, especially those involving small or mid size training databases. More importantly, and in contrast to other well known training algorithms, SVMs are very suitable when handling semi-structured and non-vectorial data (such as graphs), through the use of well dedicated kernel functions as shown subsequently.

4.1 Kernel-based Support Vector Machines

In this section, we recall the basic definitions used in kernel-based support vector machine training and show how we apply it for learning malicious behaviors. We refer the reader to (Burges, 1998) for a tutorial on this technique.

Let's consider a collection of training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$; with $\mathbf{x}_i$ being a feature in a vector space and $\mathbf{y}_i$ its class label in $\{-1, +1\}$. Support Vector Machine (SVM) training consists in finding an optimal classifier (hyperplane), denoted h, that separates labeled data in $\{(\mathbf{x}_i, \mathbf{y}_i)\}_i$ while maximizing their margin. Considering w as the normal of that

[4] Abstract API graph associated to a new program

hyperplane h, the SVM decision function of h can be written as

$$h(\mathbf{x}) = w^{\intercal}\mathbf{x} + b, \quad (1)$$

here $\mathbf{x}^{\intercal}$ stands for the transpose of $\mathbf{x}$, $w = \sum_{i=1}^{n} \alpha_i \mathbf{y}_i \mathbf{x}_i$ (with $\{\alpha_i\}_i$ being the SVM training parameters) and b is a shift. When training data are linearly separable, the hyperplane h guarantees that $\mathbf{y}_i(w^{\intercal}\mathbf{x}_i + b) \geq 1$, $\forall i \in \{1,\ldots,n\}$.

In the context of graph classification, the training set corresponds to $\{(G_i, \mathbf{y}_i)\}_i$ with G_i being an abstract API graph and $\mathbf{y}_i = +1$ if G_i is malign and $\mathbf{y}_i = -1$ otherwise. As graphs are non-vectorial data, we consider a function $\phi(.)$ which maps graphs into a high dimensional vector space (denoted $\mathcal{H}$) that also guarantees the linear separability of training data. Using ϕ, the decision function h, associated to graphs, can be written as

$$h(G) = w^{\intercal}\phi(G) + b = \sum_{i=1}^{n} \alpha_i \mathbf{y}_i \langle \phi(G_i), \phi(G) \rangle + b, \quad (2)$$

where $w = \sum_{i=1}^{n} \alpha_i \mathbf{y}_i \phi(G_i)$ and $\langle \phi(G_i), \phi(G) \rangle$ defines an inner product. Instead of ϕ, one may use the inner product $\langle \phi(G_i), \phi(G) \rangle$ and this defines a kernel function (denoted $\kappa(G_i, G)$). Conversely, a symmetric function κ defines an inner product, in some $\mathcal{H}$, iff κ is positive semi-definite(Vishwanathan et al., 2010). With this kernel definition, Equation 2 can be rewritten as

$$h(G) = \sum_{i=1}^{n} \alpha_i \mathbf{y}_i \kappa(G_i, G) + b. \quad (3)$$

Using (3) and a threshold τ, a given graph G is assigned to the malicious (resp. benign) class iff $h(G) \geq \tau$ (resp. $h(G) < \tau$) for $\tau \in \mathbb{R}$ (see Figure 4 for results w.r.t. different values of τ). The value of $h(G)$ is also seen as a confidence score of a given sample G w.r.t the positive class.

In the remainder of this section, we define the kernel function κ (used in SVMs) that implicitly maps non-vectorial data (particularly graphs) into a high dimensional vector space $\mathcal{H}$; this guarantees the linear separability of data in the mapping space $\mathcal{H}$ and also provides a relevant similarity measure between graphs in order to achieve malicious behavior detection and recognition effectively.

4.2 Random Walk Graph Kernel

Given two graphs $G = (V, E)$ and $G' = (V', E')$, the random walk graph kernel (RDW) – introduced in (Gärtner et al., 2003) – defines a similarity $\kappa(G, G')$, as the number of common walks in their product graph $G_{\times}$. The latter is a graph over pairs of vertices from G and G'; two vertices in $G_{\times}$ are connected by an edge iff the corresponding vertices in G and G' are both connected. More formally, the product graph $G_{\times} = (V_{\times}, E_{\times})$ is defined as $V_{\times} = \{(v, v') | v \in V \quad \text{and} \quad v' \in V' : \ell(v) = \ell(v')\}$ and $E_{\times} = \{((v, v'), (w, w')) | (v, w) \in E, (v', w') \in E' : \ell(v) = \ell(v') \quad \text{and} \quad \ell(w) = \ell(w')\}$, here ℓ is a labeling function[5].
With this product graph, RDW is defined as

$$\kappa(G, G') := \sum_{k=0}^{T} \mu(k) q_{\times}^{\intercal} A_{\times}^{k} p_{\times}, \quad (4)$$

here
$A_{\times}$ is the adjacency matrix of the product graph $G_{\times}$ and $A_{\times}^{k}$ is recursively defined as $A_{\times}^{k} = A_{\times}^{k-1} A_{\times}$,
$p_{\times}$ (resp. $q_{\times}$) is a vector with as many entries as vertices in G_i (resp. G_j). which characterizes the accessibility of vertices in G_i (resp. G_j). In practice, $p_{\times}$ and $q_{\times}$ are set to uniform distributions, T is the maximum length of a random walk,
$\mu(k) = \lambda^k \in [0, 1]$ is a coefficient that controls the importance of the length in random walks.
As a vertex in the product graph $G_{\times}$ corresponds to a pair of vertices (with the same API function) in the call graphs G_i, G_j, a path (with any length $k \geq 0$) in $G_{\times}$ represents a sequence of common API calls that appears in both graphs G_i and G_j; this characterizes a common behavior occurring in the two underlying programs. With this RDW kernel, the similarity between training and test data is well captured as shown through SVM classification experiments in the following section.

5 EXPERIMENTS

5.1 Dataset and Evaluation Measures

In order to evaluate the performance of our kernel-based SVMs, we collect a dataset of 6291 malware samples from Vx Heavens and 2323 benign programs from system files and applications in Windows OS and Cygwin. The proportion of malware categories is shown in Figure 2. The dataset randomly split into two partitions, a training and a testing partition. For training partition, the quantity of malwares and benign programs is balanced with 2000 samples for each. The testing set consists of 4291 malwares and 323 benign programs. In order to capture the variability of the dataset, we use 5 random splits of training and test data, then we take the average of the performances. Computing the kernel matrix for the whole

[5] In the context of the abstract API graph, the label of a vertex is an API function.

dataset of 8614 graphs takes 3 days, but this is an offline computation. Online computation to classify a new program with size 15 KB takes 15 seconds.

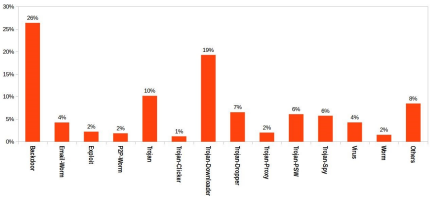


Figure 2: The malware distribution in the dataset, showing the percentage of different categories with respect to the total number of malware files.

Using this dataset, we consider two subtasks for evaluation.

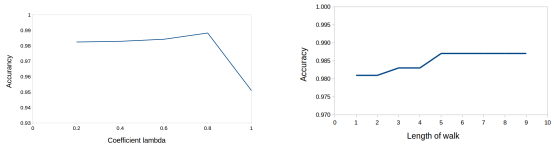
- **Malware detection.** This is the principal task of our contribution. We train a single monolithic SVM classifier (h) using positive and negative data in the training set. This classifier h is used in order to check whether a given test graph G belongs to the malign (positive) or benign (negative) class depending on the sign of $h(G)$, i.e., $\tau = 0$.
- **Malware category recognition.** As a secondary task, the goal is to recognize the category of a given malign graph G. For that purpose, we train for each category (denoted c), a "one-versus-all" SVM classifier h_c that separates graphs belonging to the c^{th} category from all others. Given a test graph G with $h(G) \geq 0$, the category of G corresponds to $\arg\max_c h_c(G)$.

In both tasks, we plug the RDW kernel in SVMs and we use the widely known library (LIBSVM)(Chang and Lin, 2011) for SVM training.

We evaluate the performance of our SVM classifiers (h and $\{h_c\}_c$) using well known measures: true positive and false positive rates respectively defined as $\text{TPR} = \text{TP}/(\text{TP}+\text{FN})$ and $\text{FPR} = \text{FP}/(\text{TN}+\text{FP})$; here TP, TN, FP, FN respectively denote true positives, true negatives, false positives and false negatives obtained after SVM classification. We also report BCR (balanced correctness rate) as one minus the average between false positive and false negative rates ($\text{BCR} = 1 - (\text{FPR}+\text{FNR})/2$) where the false negative rate $\text{FNR} = \text{FN}/(\text{FN}+\text{TN})$. Finally, we report the overall accuracy $\text{ACC} = (\text{TP}+\text{TN})/(\text{TP}+\text{TN}+\text{FP}+\text{FN})$. For all these measures, higher values of TPR, BCR, ACC (with small values of FPR) imply better performances.

5.2 Performances and Comparison

Performances of Malware Detection. Firstly, we measure the performance of the RDW kernel (combined with SVM) w.r.t different walk lengths (i.e., w.r.t parameter T in Eq. 4) and different values of coefficient λ to control the importance of the length in random walks in $\mu(k) = \lambda^k$ of Eq. 4. Figure 3(a) shows that the classification accuracy ACC increases as λ increases from 0.2 to 0.8, then it decreases after reaching the max value at $\lambda = 0.8$. Figure 3(b) shows that as T increases, the classification accuracy ACC increases and stabilizes when T reaches 5 random walks. Following these results, T is fixed to 5 and λ is fixed to 0.8 in all the remaining experiments; with this setting, detection rates (TPR), reported in Table 1 reach 98.93%, with a false positive rate FPR of 1.24%.



(a) (b)

Figure 3: This diagram shows the evolution of the Accuracy w.r.t λ (a) and T (b) in the RDW kernel.

Table 1: This table shows the performances of RDW kernel. We obtain these results by averaging the results of 5 runs, each run corresponds to a random split of the dataset into training and test data.

TP	TN	FP	FN	TPR	FPR	ACC
4245	319	4	46	98.93%	1.24%	98.91%

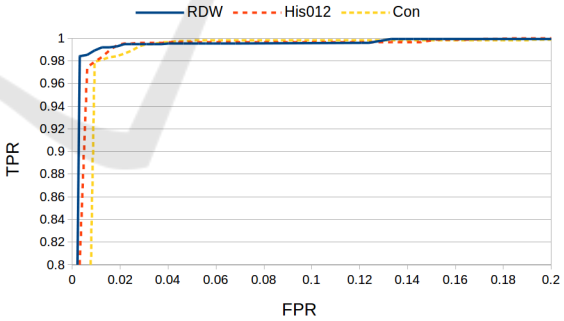


Figure 4: This figure shows true positive rates vs. false positive rates of our method and its comparison against the two baseline kernels: structured histogram intersection and convolution kernels (referred to as His012 and Con). These results show that RDW achieves the best true positive rate (around 99% with a small FPR; around 1%) compared to histogram intersection and convolution kernels (which achieve a TPR of 98%).

Secondly, we compare the performance of the RDW kernel against two widely used baseline kernels for graph comparison: (i) convolution kernel and (ii) structured histogram intersection kernel. Given two graphs $G = (V,E)$ and $G' = (V',E')$, the convolu-

tion kernel introduced in (Haussler, 1999) for semi-structured data (including graphs), is defined as $\kappa(G,G') = \frac{1}{|V|\times|V'|}\sum_{v\in V}\sum_{v'\in V'} 1_{\{\ell(v)=\ell(v')\}}$, here $1_{\{\}}$ corresponds to the indicator function.
The second baseline kernel – structured histogram intersection – is defined as

$$\kappa(G,G') = \kappa_0(G,G') + \kappa_1(G,G') + \kappa_2(G,G'), \quad (5)$$

here $\kappa_0(G,G')$, $\kappa_1(G,G')$ and $\kappa_2(G,G')$ correspond to standard histogram intersection kernels associated to cliques of order 0, 1 and 2 respectively (i.e., vertices, edges and connected subgraphs with 3 vertices). Following (Barla et al., 2003; Maji et al., 2008), these three kernels are defined as

$$\begin{aligned}
\kappa_0(G,G') &= \textstyle\sum_{i=1}^{L} \min(g_0(G,\ell_i), g_0(G',\ell_i)) \\
\kappa_1(G,G') &= \textstyle\sum_{i,j=1}^{L} \min(g_1(G,\ell_i,\ell_j), g_1(G',\ell_i,\ell_j)) \\
\kappa_2(G,G') &= \textstyle\sum_{i,j,k=1}^{L} \min(g_2(G,\ell_i,\ell_j,\ell_k), g_2(G',\ell_i,\ell_j,\ell_k)),
\end{aligned} \quad (6)$$

here L is $|\mathcal{A}|$, i.e., is the number of API functions in the program. $g_0(G,\ell_i)$ is the probability of occurrence of label ℓ_i in G, i.e., $g_0(G,\ell_i) = \frac{1}{|V|}\sum_{v\in V} 1_{\{\ell(v)=\ell_i\}}$. Similarly, $g_1(G,\ell_i,\ell_j)$ (resp. $g_2(G,\ell_i,\ell_j,\ell_k)$) corresponds to the probability of occurrence of edges with labels (ℓ_i,ℓ_j) (resp. connected triplet of vertices with labels (ℓ_i,ℓ_j,ℓ_k)).
Figure 4 shows the evolution of the true positive rate (TPR) and the false positive rate (FPR) w.r.t. different and increasing values of τ, taken from min to max value of $h(G)$, i.e., $\tau \in [-4,6]$. The interesting part of these diagrams corresponds to small values of FPR; indeed, for reasonably small and comparable FPRs, our method based on the RDW kernel has high TPRs and it clearly overtakes the convolution kernel as well as structured histogram intersection kernel.
Performances of Malware Category Recognition. Again, given a graph G (with $h(G) \geq 0$), the goal is to assign it to one of 13 malware categories (Backdoor, Email-Worm, Exploit, P2P-Worm, Trojan, Trojan-Clicker, Trojan-Downloader, Trojan-Dropper, Trojan-Proxy, Trojan-PSW, Trojan-Spy, Virus and Worm) based on $\arg\max_c h_c(G)$ (thanks to Equation 3). Figure 5 shows the classwise TPR, Accuracy and BCR rates of our RDW kernel and its comparison against the two other baseline kernels.
Comparison with Well-known Antiviruses. We compare the performance of our method with different existing antiviruses including Avira, Kaspersky, Avast, Qihoo-360, McAfee, AVG, BitDefender, ESET-NOD32, F-Secure, Symantec and Panda. Since known antiviruses update their signature database as soon as a new malware is known, in order to have a fair comparision with these antiviruses, we need to consider new malwares. For this, we use three generators to create new malwares: NGVCK, RCWG and

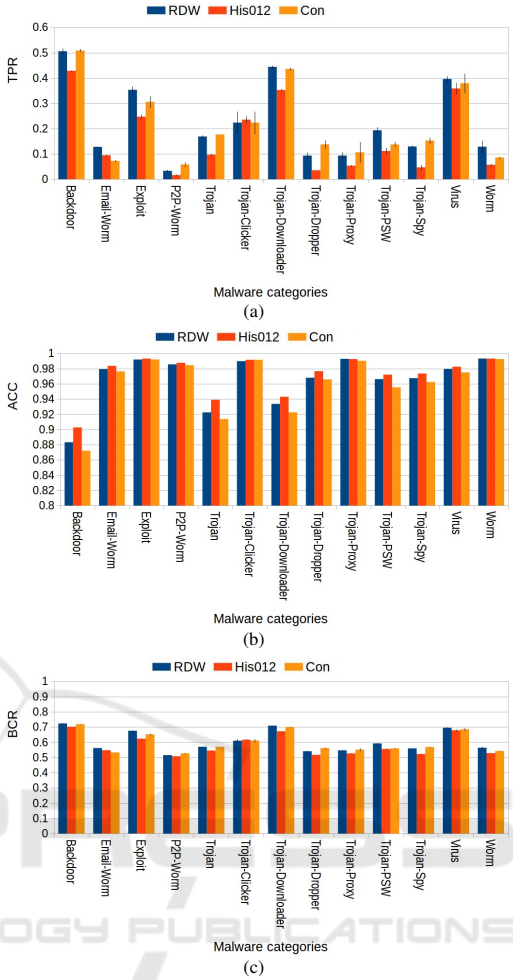


Figure 5: This figure shows class-by-class malware category recognition performances of our RDW-based method and the two other baselines kernels on 13 malware categories. Fig. (a), (b) and (c) show detection rate (TPR), accuracy and balanced correctness rate (BCR) for each malware category and their variances. Our method reaches a BCR of 60% while histogram intersection and convolution kernels obtain 58% and 59% respectively. Results are averaged over five experimental runs.

VCL32. The latter are able to create sophisticated malwares with morphing code and other features to avoid being detected by antiviruses. In total, we generate 180 new malwares by RCWG, VCL32 and NGVCK generators. After training our SVM classifier on the training set, we are able to detect 100% of new malwares while none of the well known antiviruses can detect all of them. The results are shown in Table **??**.

Table 2: This table shows a comparison of our method against well-known antiviruses. Our tool achieves a detection rate of **100%**.

Antivirus	Detection Rates	Antivirus	Detection Rates
Our tool	**100%**	Panda	19%
Avira	16%	Kaspersky	81%
Avast	87%	Qihoo-360	96%
McAfee	96%	AVG	82%
BitDefender	87%	ESET-NOD32	87%
F-Secure	87%	Symantec	14%

6 CONCLUSION

The main contribution of this paper is the application of graph kernel based learning techniques for malware detection in a completely static way (no dynamic analysis). As far as we know, this is the first time that these techniques are applied for malware detection in a static manner. We introduced an automatic malware detection algorithm based on SVMs. First, we use static analysis in order to create abstract API graphs from control flow graphs. Then, we build SVMs that learn the malicious behaviors from these API graphs and achieve malware detection and recognition. These SVMs are built upon a well dedicated random walk graph kernel (RDW) that measures graph similarity as the number of common paths of increasing lengths and characterizes common malicious behaviors through training and test data. The use of this kernel is clearly appropriate as it allows us to handle non-vectorial data (i.e., graphs) without any explicit generation of features on these graphs. Experiments show that our RDW-based classifier achieves a TPR of almost 99% with only 1.24% FPR for malware detection and an accuracy of 96.55% for malware category recognition. Compared to other kernels (such as histogram intersection and convolution), our RDW based method obtains the best classification performances.

Note that we could have extracted vectorial features from graphs and then applied other learning techniques such as ANNs, but this would have led to loss of information. Thus, we believe that applying graph kernel based SVMs is the best choice to learn our malicious behavior graphs.

REFERENCES

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

Babić, D., Reynaud, D., and Song, D. (2011). Malware analysis with tree automata inference. CAV'11.

Barla, A., Odone, F., and Verri, A. (2003). Histogram intersection kernel for image classification. In *ICIP 2003*.

Bergeron, J., Debbabi, M., Erhioui, M., and Ktari, B. (1999). Static analysis of binary code to isolate malicious behaviors. In *WET ICE '99*.

Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2).

Chang, C.-C. and Lin, C.-J. (2011). Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

Christodorescu, M. and Jha, S. (2003). Static analysis of executables to detect malicious patterns. SSYM'03.

Christodorescu, M., Jha, S., and Kruegel, C. (2007). Mining specifications of malicious behavior. ESEC-FSE '07. ACM.

Eagle, C. (2011). *The IDA Pro Book*. No Starch Press, 2nd edition.

Elhadi, E., Maarof, M. A., and Barry, B. (2015). Improving the detection of malware behaviour using simplified data dependent api call graph.

Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., and Yan, X. (2010). Synthesizing near-optimal malware specifications from suspicious behaviors. SP '10.

Gärtner, T., Flach, P., and Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*.

Gavrilut, D., Cimpoesu, M., Anton, D., and Ciortuz, L. (2009). Malware detection using perceptrons and support vector machines. In *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. IEEE.

Haussler, D. (1999). Convolution kernels on discrete structures.

Khammas, B. M., Monemi, A., Bassi, J. S., Ismail, I., Nor, S. M., and Marsono, M. N. (2015). Feature selection and machine learning classification for malware detection. *Jurnal Teknologi*, 77.

Kinable, J. and Kostakis, O. (2011). Malware classification based on call graph clustering. *J. Comput. Virol.*, 7(4).

Kinder, J., Katzenbeisser, S., Schallhart, C., and Veith, H. (2010). Proactive detection of computer worms using model checking. *Dependable and Secure Computing, IEEE Transactions on*, 7(4).

Kinder, J. and Veith, H. (2008). Jakstab: A static analysis platform for binaries. In Gupta, A. and Malik, S., editors, *Computer Aided Verification*, volume 5123.

Kolter, J. Z. and Maloof, M. A. (2004). Learning to detect malicious executables in the wild. KDD '04.

Kong, D. and Yan, G. (2013). Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Macedo, H. and Touili, T. (2013). Mining malware specifications through static reachability analysis. In *ESORICS 2013*.

Maji, S., Berg, A., and Malik, J. (2008). Classification using intersection kernel support vector machines is efficient. In *CVPR 2008*.

Nguyen, M. H., Nguyen, T. B., Quan, T. T., and Ogawa, M. (2013). A hybrid approach for control flow graph construction from binary code. In *APSEC 2013*, volume 2.

Nikolopoulos, S. D. and Polenakis, I. (2016). A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques*, pages 1–18.

Ravi, C. and Manoharan, R. (2012). Malware detection using windows api sequence and machine learning. *International Journal of Computer Applications*, 43.

Rieck, K., Holz, T., Willems, C., Dussel, P., and Laskov, P. (2008). Learning and classification of malware behavior. DIMVA '08.

Schultz, M., Eskin, E., Zadok, E., and Stolfo, S. (2001). Data mining methods for detection of new malicious executables. In *S P 2001*.

Song, F. and Touili, T. (2013a). Ltl model-checking for malware detection. In Piterman, N. and Smolka, S., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795.

Song, F. and Touili, T. (2013b). Pommade: Pushdown model-checking for malware detection. ESEC/FSE 2013.

Tahan, G., Rokach, L., and Shahar, Y. (2012). Mal-id: Automatic malware detection using common segment analysis and meta-features. *J. Mach. Learn. Res.*, 13(1).

Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., and Borgwardt, K. M. (2010). Graph kernels. *J. Mach. Learn. Res.*, 11.

Wagner, C., Wagener, G., State, R., and Engel, T. (2009). Malware analysis with graph kernels and support vector machines. In *MALWARE 2009*. IEEE.

Xu, M., Wu, L., Qi, S., Xu, J., Zhang, H., Ren, Y., and Zheng, N. (2013). A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques*, 9(1):35–47.