# Learning Malware Using Generalized Graph Kernels

Khanh Huu The Dam
LIPN, CNRS and University Paris 13, France

Tayssir Touili
CNRS, LIPN and University Paris 13, France

## ABSTRACT

Machine learning techniques were extensively applied to learn and detect malware. However, these techniques use often rough abstractions of programs. We propose in this work to use a more precise model for programs, namely *extended API call graphs*, where nodes correspond to API function calls, edges specify the execution order between the API functions, and edge labels indicate the dependence relation between API functions parameters. To learn such graphs, we propose to use Generalized Random Walk Graph Kernels (combined with Support Vector Machines). We implemented our techniques and obtained encouraging results for malware detection: 96.73% of detection rate with 0.73% of false alarms.

## KEYWORDS

Malware detection, Support Vector Machine, Graph Kernel

**ACM Reference Format:**
Khanh Huu The Dam and Tayssir Touili. 2018. Learning Malware Using Generalized Graph Kernels. In *ARES 2018: International Conference on Availability, Reliability and Security, August 27–30, 2018, Hamburg, Germany.* , 6 pages. https://doi.org/10.1145/3230833.3230840

```
n1 : push FindFileData          n9 : call MoveFile
n2 : push ".exe"                n10 : push 0
n3 : call FindFirstFile         n11 : lea eax, [edi + 2Ch]
n4 : push eax                   n12 : push eax
n5 : mov edi, FindFileData      n13 : mov eax, ExistingFile
n6 : lea eax, [edi + 2Ch]       n14 : push eax
n7 : push NewFileName           n15 : call CopyFile
n8 : push eax
```

(a)

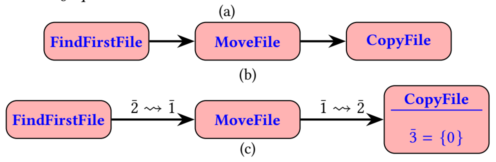

**Figure 1: A piece of assembly code (a) of the behavior of infecting files in the system, its API call graph (b) and its extended API call graph (c).**

## 1 INTRODUCTION

Malware detection is nowadays a big challenge. Indeed, there are more than 600 millions of new variants of malwares released in 2017 [23]. However, specifying the malicious behaviors currently requires a huge amount of engineering effort and an enormous amount of time. Thus, the main challenge is how to avoid the step of manually reading the code to discover the malicious behaviors. For this purpose, machine learning techniques can be applied to *automatically* classify malware. A lot of works have been proposed in this direction (see the related work section for more details). However, the majority of these works model a program as a linear vector, and apply machine learning techniques on these vectors. Such techniques are not very precise as they perform a rough abstraction by representing a program by a linear vector.
To consider more precise models of programs, it has been widely observed that malicious tasks are usually performed by calling sequences of API functions, since API functions allow to access the system and modify it. Thus, in previous works [9, 10, 15, 18, 32], the malicious behaviors were characterized as API call graphs where nodes are API functions. Such graphs represent the execution order of API function calls in the program. For instance, let us look at a typical malicious behavior implemented by the assembly code in Figure 1, where the malware searches for executable files in the system and replaces them by the malicious program. This is achieved by first calling the API function `FindFirstFile` to search the executable files in the system. Then, the malware calls the function `MoveFile` with the second parameter (i.e., the output) of `FindFirstFile` as first parameter (Parameters to a function in assembly are passed by pushing them onto the stack before a call to the function is made. The code in the called function later retrieves these parameters from the stack.) to move this executable file to a new location or change its name. Finally, the API function `CopyFile` with the first parameter of `MoveFile` as second parameter and 0 as third parameter is called to replace this executable file in the current searching location by the malicious program. To represent this behavior, [9, 10, 15, 18, 32] use the API call graph in Figure 1(b) to express that calling `FindFirstFile` is followed by a call to `MoveFile` and calling `MoveFile` is followed by a call to `CopyFile`. However, a program that contains this behavior is malicious only if the API function `MoveFile` moves the file returned by `FindFirstFile` and the API function `CopyFile` copies the file used by `MoveFile`. If `MoveFile` and `CopyFile` move/copy another file, the behavior is not malicious. Thus, the above representation may lead to false alarms. To avoid this, we need to make the representation more precise and add the information that the returned parameter of `FindFirstFile` should be the input argument of `MoveFile` and `CopyFile`. Therefore, we propose to use the Extended API call graph in Figure 1(c), where the edge labeled by $\bar{2} \rightsquigarrow \bar{1}$ means that the second parameter of `FindFirstFile` (which is its output) is given as first argument of `MoveFile` and the other edge labeled by $\bar{1} \rightsquigarrow \bar{2}$ means that the first parameter of `MoveFile` (which gets the output of `FindFirstFile`) is given as second parameter of `CopyFile`. We also need to ensure that `CopyFile` is called with 0 as third parameter. Thus, we label the node `CopyFile` with $\bar{3} = \{0\}$ to express that the third parameter of this call should be 0. Thus, we propose in this work to use *Extended API call graphs* to represent malicious behaviors.

*ARES 2018, August 27–30, 2018, Hamburg, Germany*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6448-5/18/08…$15.00
https://doi.org/10.1145/3230833.3230840

An extended API call graph is a directed graph whose nodes are API functions. An edge $(f, f')$ expresses that there is a call to the API function $f$ followed by a call to the API function $f'$. The annotation $\bar{i} \rightsquigarrow \bar{j}$ on the edge $(f, f')$ means that the $i^{th}$ parameter of function $f$ and the $j^{th}$ parameter of the function $f'$ have a data dependence relation. It means that, in the program, either these two parameters depend on the same value or one parameter depends on the other, e. g., in Figure 1(c) the edge (`FindFirstFile`, `MoveFile`) with label $\bar{2} \rightsquigarrow \bar{1}$ expresses that the second parameter ($\bar{2}$) of `FindFirstFile` and the first parameter ($\bar{1}$) of `MoveFile` gets the same memory address $FindFileData$. A node $f$ with the annotation $\bar{i} = \{c\}$ means that the $i^{th}$ parameter of function $f$ gets the value $c$, e. g., the node `CopyFile` in Figure 1(c) is associated with the label $\bar{3} = \{0\}$. This graph specifies the execution order of the API function calls like the API call graph in [9, 10, 15, 18, 32]. In addition, it records the links between the API functions' parameters.

Using this representation, given a set of extended API call graphs that correspond to malwares and a set of extended API call graphs corresponding to benign programs, our goal is to apply machine learning techniques to *automatically* learn the malicious behaviors. We apply kernel based support vector machines (SVMs) to compute an optimal classifier which is used to classify new programs. Indeed, SVMs are shown to be very successful when applied in many pattern classification problems, especially those involving small or mid size training databases. In contrast to other well known learning algorithms like artificial neural network, k-nearest neighbor, decision trees, etc., SVMs are very suitable when handling semi-structured and non-vectorial data (such as graphs), through the use of well dedicated kernel functions. To apply kernel based SVMs, we first need to define an adequate kernel, i. e., a function that computes the similarity between graphs. To compute similarities between standard API call graphs, [10] use random walk graph kernels that measure graph similarities as the number of common walks of increasing lengths. Such kernel cannot be used in our context since we have extended API call graphs, where each node/edge is associated with a label. To overcome this problem, we propose to adapt the *generalized* random walk graph kernels defined in [30] to compute similarities between extended API call graphs, and learn malware. As far as we know, this is the first time that the *generalized* random walk graph kernels are used for malware learning and detection.

We implement our techniques in a tool and test it on a dataset of 4035 malwares that are collected from Vx Heaven [7] and from VirusShare [1] and 2249 benign programs. Our experimental results show that our generalized graph kernel is able to capture well the structure of extended API call graphs since it leads to a detection rate of 97% with a low false alarm of 0.73%. These results outperform the ones of [10] obtained using standard API call graphs with the standard random walk graph kernel.

**Outline.** The next section describes the related work. Section 3 presents the definition of extended API call graphs. In Section 4, we present an adaptation of the *generalized* random walk graph kernels for our extended API call graphs and show how this can be used to learn malware. The experimental results are reported in Section 5.

## 2 RELATED WORK

Machine learning techniques were applied for malware classification in [13, 14, 17, 26, 27]. However, all these works use either a vector of bits [13, 26] or n-grams [14, 17, 27] 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 evaded 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.

[22] uses 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. Similarly, [19] also takes into account the system calls in the program. However, the authors only consider the length of the string arguments and the distribution of characters in the string arguments as features for their learning models. [24] 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 representations as it allows to take into account several API function sequences in the program while keeping the order of their execution. Moreover, [22], [19] and [24] use dynamic analysis to extract a program's representation.

[11, 15, 18, 21, 32] use graphs where nodes are functions of the program (either API functions or any other function of the program). [11] 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. [15] 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. [18] applies standard support vector machines, where instead of using graph kernels to compute similarity between graphs, the authors abstract graphs by a linear vector and compute similarity by comparing these vectors. Our graph kernel approach is more precise since it is more suitable for graphs. [32] and [21] classify malwares according to a similarity metric computed by a kind of graph matching while our classifier is built from SVM classifier with the random walk graph kernel. Moreover, [21] uses a kind of API call graph, where each node corresponds to a group of API function calls. Our graphs are more precise since we do not group API functions together.

Graph kernel based SVM for malware detection is used in [2, 10, 29]. [29] 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, [29] 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 [2], 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, [2] 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.
[10] applies the standard random walk graph kernel to compute the similarity between two API call graphs. However, this graph kernel is not suitable for our extended API call graphs since it cannot capture well the labels on both edges and nodes in the computation. Moreover, these graphs used in [10] are less precise than our extended API call graphs, as they consider only the execution order between API functions, they do not consider neither the parameter values, nor the data dependence between the functions' arguments. In Section 5, we give experimental evidence that shows that our approach is better than the one of [10].

Extended API call graphs were introduced in [8], where static analysis techniques that allow to automatically and statically extract extended API call graphs from binary programs are presented. This work does not consider learning of malware. Instead, given a set of extended API call graphs corresponding to malicious and benign programs, the authors apply Information retrieval techniques based on the TFIDF schema to extract an extended API call graph that corresponds to the malicious behaviors of these malwares. In [6, 11, 12, 16, 20, 21], the authors represent programs using graphs similar to our extended API call graphs. However, these works [6, 11, 12, 16] use dynamic analysis to compute the graphs, whereas our graph representation is made statically. Static analysis is much more precise as it allows to consider all programs paths while dynamic analysis considers only a finite number of paths of the program. In [20], the authors try to use static analysis to model the graphs corresponding to the malicious behaviors. However, their representation is not precise, as they state that two variables are related if they have the same value, not if they depend on each other.

## 3 EXTENDED API CALL GRAPHS

Let $\mathcal{F}$ be the set of all API functions that are called in the program. For every API function $f \in \mathcal{F}$, let $Para(f)$ be the set of parameters of $f$ and $|Para(f)|$ be the number of parameters of $f$. For each API function, there can be special parameters on which the behavior (the output) of the API function depends, e. g., calling the API function `CopyFile` with 0 as third parameter allows to overwrite the current file by the malicious program, thus the value of the third parameter is crucial for the behavior of the API function `CopyFile`. We call such parameters *meaningful* parameters. Let $Para_M(f)$ be such *meaningful* parameters of $f$.

An extended API call graph is a directed graph $G = (V, E)$ such that: $V = V_1 \cup V_2$, where $V_1 \subseteq \{(f, eval) \mid f \in \mathcal{F}, Para_M(f) \neq \emptyset$, and $eval$ is a function $eval : Para_M(f) \to 2^{\mathbb{Z}}\}$ is the set of vertices consisting of pairs of the form $(f, eval)$ for an API function $f$ and an evaluation $eval$ that specifies the value of the meaningful parameters of $f$, and $V_2 \subseteq \{f \mid f \in \mathcal{F}, Para_M(f) = \emptyset\}$ is the set of vertices labelled by API functions with no meaningful parameter.

Let $v \in V$. Let $f \in \mathcal{F}$ be such that $v$ is of the form $(f, eval)$ or $f$. Then, we define $func(v) = f$ and $Para(v) = Para(f)$. Moreover, if $v$ of the form $(f, eval)$, then $mean(v) = eval$, and if $v$ of the form $f$, then $mean(v) = \emptyset$.

$E \subseteq \{(v_1, 2^{|Para(v_1)| \times |Para(v_2)|}, v_2) \mid v_1 \text{ and } v_2 \in V\}$ is the set of edges. $(v_1, e, v_2) \in E$ means that the API function $func(v_1)$ with meaningful parameter values defined by $mean(v_1)$ is called before the API function $func(v_2)$ with meaningful parameter values defined by $mean(v_2)$. Moreover, $(i, j) \in e$ means that the $i^{th}$ parameter of $func(v_1)$ and the $j^{th}$ parameter of $func(v_2)$ are related.

For example, let us consider the extended API call graph of Figure 1(c). Let $a_3$ represent the third argument of function `CopyFile`. Here, $a_3$ is meaningful, whereas `FindFirstFile` and `MoveFile` do not have any meaningful parameter. Thus, $V = \{$ (`CopyFile`, $eval$), `FindFirstFile`,`MoveFile` $\mid eval(a_3) = \{0\}\}$: $eval(a_3) = \{0\}$ expresses that when `CopyFile` is called, the third parameter has to be equal to 0. $E = \{($ `FindFirstFile`, $(2,1)$ , `MoveFile` $), ($ `MoveFile`, $(1,2)$, (`CopyFile`, $eval$)$)\}$: the pair $(2,1)$ expresses that the second parameter of `FindFirstFile` serves as first parameter of `MoveFile` and the pair $(1,2)$ expresses that the first parameter of `MoveFile` serves as second parameter of `CopyFile`.

**Graphical representation.** Note that in our graphical representation in Figure 1(c), $\bar{3} = \{0\}$ represents $eval(a_3) = \{0\}$, whereas $\bar{1} \rightsquigarrow \bar{2}$ and $\bar{2} \rightsquigarrow \bar{1}$ stand for $(1,2)$ and $(2,1)$, respectively.

**Computing the extended API call graph of a binary program.** Given a binary program, we use the techniques of [8] to automatically and statically compute an extended API call graph corresponding to a binary program. This task is complex, since in assembly, parameters are passed through the stack. Therefore, to determine the relations between functions' parameters, we need to access the program's stack. To tackle this problem, we model a binary program using a Pushdown system (PDS), we represent potentially infinite sets of configurations of PDSs using finite state automata, and we extend the $post^*$ saturation procedure of [3] in order to compute the dependence relations beteween variables and stack parameters. More details about our technique can be found in [8].

## 4 LEARNING MALICIOUS BEHAVIORS

In this section, we recall the basic definitions used in kernel-based support vector machine training and show how we adapt it to learn extended API call graphs for malware detection.

### 4.1 Kernel Based Support Vector Machines

Let us consider a collection of training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$; with $\mathbf{x_i}$ being a feature in a vector space and $y_i$ its class label in $\{-1, +1\}$. The support vector machines technique consists in finding a hyperplane $h$ that separates these data $\mathbf{x}_i$ $(1 \le i \le n)$ while maximizing their margin [4, 25]. Then, given a new unseen data $\mathbf{x}' \in X$, the SVM decision function can be written as

$$h(\mathbf{x}') = \mathbf{w}^\top \mathbf{x}' + b. \tag{1}$$

where $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$ with $\{\alpha_i\}_{1 \le i \le n}$ being the SVM training parameters and $b$ is a shift. When training data are linearly separable, the hyperplane $h$ guarantees that $y_i(\mathbf{w}^\top \cdot \mathbf{x}_i + b) \ge 1, \forall i \in \{1, \ldots, n\}$. In the context of graph classification, the training set corresponds to $\{g_i, y_i\}_i$ with $g_i$ being an extended API call graph and $y_i = +1$ if $g_i$ is malicious and $y_i = -1$ otherwise. Since our extended API call graphs are not vectorial data, we consider a mapping function $\phi(.)$ which maps these graphs into a high dimensional vector space (denote $\mathcal{H}$) that guarantees the linear separability of the training data. Using the mapping function $\phi$, the decision function $h$ for a given graph $g'$ can be written as

$$h(g') = \mathbf{w}^\intercal \phi(g') + b = \sum_{i=1}^{n} \alpha_i y_i \left\langle \phi(g_i) \cdot \phi(g') \right\rangle + b \qquad (2)$$

where $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \phi(g_i)$ and $\langle \phi(g_i) \cdot \phi(g') \rangle$ defines an inner product. Instead of using $\phi(.)$, one may use the inner product $\langle \phi(g_i) \cdot \phi(g') \rangle$ and this defines a kernel function $\kappa(g_i, g')$ [31]. Conversely, a symmetric function $\kappa$ defines an inner product, in some vector space $\mathcal{H}$, iff $\kappa$ is positive semi-definite [28]. If we replace the inner product $\langle \phi(g_i) \cdot \phi(g') \rangle$ by a kernel function $\kappa(g_i, g')$, Equation 2 can be rewritten as

$$h(g') = \sum_{i=1}^{n} \alpha_i y_i \kappa(g_i, g') + b \qquad (3)$$

Using Equation 3 and a threshold $\tau \in \mathbb{R}$, for any given extended API call graph $g'$ if $h(g') > \tau$, $g'$ is marked malicious. Otherwise, $g'$ is marked benign. The value of $h(g')$ is also seen as a confidence score of a given graph $g'$ w.r.t the positive class.

## 4.2 Similarity between nodes and edges

The kernel based support vector machine technique described in the previous section is based on the definition of the kernel function $\kappa$ to compare graphs. The choice of the kernel $\kappa$ is crucial for the efficiency of this technique. This kernel measures the similarity between graphs. Thus, to define $\kappa$, we first need to define a similarity measure between edges and nodes. In extended API call graphs, each node/edge is associated with a label. Thus, the similarity between nodes and edges has to compare these labels. First, given two extended API call graphs $g = (V, E)$ and $g' = (V', E')$, we define the extended product graph $g_\times = (V_\times, E_\times)$ as follows: $V_\times = \{(v, v') | v \in V \text{ and } v' \in V' : func(v) = func(v')\}$ and $E_\times = \{\big((v_1, v_1'), (v_2, v_2')\big) | (v_1, \ell, v_2) \in E, (v_1', \ell', v_2') \in E' : (v_1, v_1')$ and $(v_2, v_2') \in V_\times\}$. Each node $(v, v')$ of the extended product graph $g_\times$ is associated with a weight specifying the similarity of the labels of $v$ and $v'$. Let $(v, v')$ be a node in $V_\times$. Let $f$ be the corresponding API function, i. e., $f = func(v) = func(v')$. The similarity of two nodes $v \in V$ and $v' \in V'$ is measured as follows:

$$Sim_{nodes}(v, v') = \frac{|Para_\times(v, v')| + 1}{|Para_M(f)| + 1}$$

where $Para_\times(v, v') = \{a | a \in Para_M(f) : mean(v)(a) = mean(v')(a)\}$ is the set of meaningful parameters of $f$ that have the same values at $v$ and $v'$. $|Para_M(f)|$ is the total number of meaningful parameters of function $f$. Intuitively, the similarity of $v$ and $v'$ is the number of meaningful parameters that have the same values at $v$ and $v'$. In order to make the similarity values comparable between different API functions having different numbers of meaningful parameters, we normalize this similarity value by dividing by the number of meaningful parameters. Moreover, since the functions may have no parameter, i. e., $|Para_M(f)| = 0$, we add the coefficient 1 to avoid having a problem in this case.

Each edge $((v_1, v_1'), (v_2, v_2'))$ of the product graph $g_\times$ is associated with a weight specifying the number of the common labels of edges $e = (v_1, \ell, v_2)$ of the graph $g$ and $e' = (v_1', \ell', v_2')$ in the graph $g'$. The weight associated to a pair of edges in the product graph is computed as follows.

$$Sim_{edges}(e, e') = \frac{1}{2}\Big(\frac{|\ell_\times| + 1}{|\ell| + 1} + \frac{|\ell_\times| + 1}{|\ell'| + 1}\Big)$$

where $\ell_\times = \{(i, j) | (i, j) \in \ell \text{ and } (i, j) \in \ell'\}$ is the set of common labels of $e$ and $e'$. $|\ell|$ and $|\ell'|$ are the number of labels at $e$ and $e'$, respectively. Intuitively, the weight of a pair of edges in the extended product graph is the average of the number of common labels over the number of labels on each edge in this pair. We divide by the number of labels of each edge in the pair $(e, e')$ in order to make this value comparable between pairs of edges in the product graph. Since the edge may have no label, i. e., $|\ell| = 0$ or $|\ell'| = 0$, we add the coefficient 1 to avoid having a problem in this case.

## 4.3 Generalized random walk graph kernel

We are now ready to define our kernel function $\kappa$. Given an extended product graph $g_\times = (V_\times, E_\times)$ of two extended API call graphs $g = (V, E)$ and $g' = (V', E')$, following [30], we define the generalized random walk graph kernel as

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

where $p_\times$ (resp. $q_\times$) is a vector with as many entries as nodes in $g$ (resp. $g'$) which characterizes the accessibility of nodes in $g$ (resp. $g'$). 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.
$A_\times$ is the matrix s.t. for every $i = (v_1, v_1') \in V_\times$ and $j = (v_2, v_2') \in V_\times$

$$A_\times[i, j] = \begin{cases} Sim_{edges}(e, e') & \text{if} \quad e = (v_1, \ell, v_2) \in E \text{ and} \\ & \qquad e' = (v_1', \ell', v_2') \in E', \\ 0 & \qquad otherwise. \end{cases}$$

and $W$ is a diagonal matrix s.t. for every $i = (v, v') \in V_\times$

$$W[i, i] = Sim_{nodes}(v, v').$$

This kernel allows to capture the similarity between two extended API call graphs according to the similarity between edges and nodes defined above. Then, we can plug this kernel into Support Vector Machine (SVM) in order to perform extended API call graph (malware) classification. As far as we know, this is the first time that generalized random walk graph kernels (first defined in [30]) are used for malware learning and detection.

**Intuition.** This kernel computes the similarity between two extended API call graphs by summing up the similarity between walks of increasing lengths in these graphs. As a node in the product graph $g_\times$ corresponds to a pair of nodes (with the same API function) in the graphs $g$ and $g'$, a path (with any length $k \geq 0$) in $g_\times$ represents a sequence of common API calls that appears in both graphs $g$ and $g'$. Besides, the weight of nodes in the product graph $g_\times$ is the similarity of two nodes (two API function calls with their arguments) in the graphs $g$ and $g'$. This similarity is taken into account in the generalized random walk graph kernel by the diagonal matrix $W$. Similarly, the weight matrix $A_\times$ represents the similarity of a pair of edges in the graphs $g$ and $g'$. Thus, $W[i, i]A_\times[i, j]W[j, j]$ is the similarity of walks of length 1 connecting node $i$ and $j$ in the product graph. Similarly, $\sum_k W[i,i]A_\times[i,k]W[k,k]A_\times[k,j]W[j,j]$ is the sum of similarities of all walks of length 2 starting from node $i$ and ending at node $j$. Thus, if $K^t$ denotes $(WA_\times)^t W$, $\sum_k K^{(t-1)}[i,k]A_\times[k,j]W[j,j]$ is the sum of all the similarities of all walks of length $t$ from node $i$ to node $j$.

## 5 EXPERIMENTS

To evaluate the performance of our approach, we implement it and evaluate it on a dataset of 2249 benign programs and 4035 malicious programs collected from Vx Heaven [7] and from VirusShare.com [1]. 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 1009 samples for each, that will allow us to compute the SVM classifier. The testing set consists of 3026 malwares and 1240 benign programs, that is used to evaluate the classifier.

As mentionned previously, [10] model programs as standard API call graphs (where edges are not labelled and nodes have only API functions as labels), and apply standard random walk (RDW) graph kernels on these standard API call graphs to learn malware. To compare the technique of [10] with ours, we implement it and compare its efficency to ours on the same dataset of malicious and benign programs.

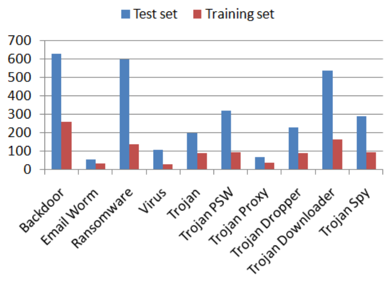

**Figure 2: Malware categories in our dataset.**

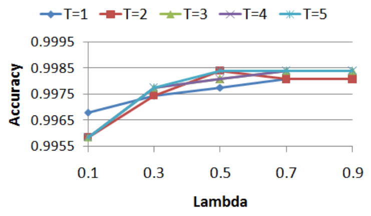

**Figure 3: The evolution of the Accuracy on the training set w.r.t $\lambda$ and $T$ in the kernel function.**

Using this dataset, we consider two subtasks:

– **Malware detection.** We train a binary 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 $c^{\text{th}}$ 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 generalized random walk graph (GRDW) kernel in SVMs and use the library (LIBSVM) [5] for SVM training. Besides, for each task we compare the results we obtain using our techniques against the ones of [10] obtained using API call graphs and the RDW kernel.

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 TPR = TP/(TP + FN) and FPR = FP/(TN + 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 (BCR = 1 − (FPR + FNR)/2) where the false negative rate FNR = FN/(FN+TN). Finally, we report the overall accuracy ACC = (TP + TN)/(TP + TN + FP + FN). For all these measures, higher values of TPR, BCR, ACC (with small values of FPR) imply better performances.

| | TP | TN | TPR (%) | FPR (%) | ACC (%) |
|---|---|---|---|---|---|
| Extended API call graphs | 2927 | 1231 | 96.73 | 0.73 | 97.47 |
| API call graphs | 2807 | 1199 | 92.76 | 3.31 | 93.91 |

**Table 1: The performance of malware classification on our extended API call graphs with the GRDW kernel v.s. API call graphs with the RDW kernel.**

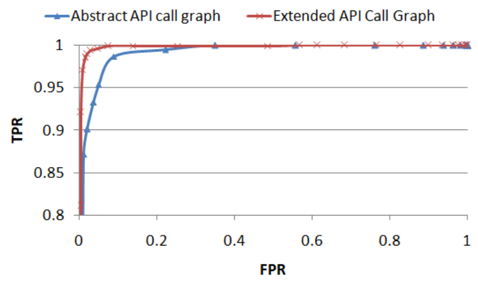

**Figure 4: True positive rates vs. False positive rates for the two approaches.**

**Performances of Malware Detection.** First, we measure the performance of the GRDW kernel (combined with SVM) w.r.t different walk lengths (i.e., w.r.t parameter $T$ in Figure 4) and different values of coefficient $\lambda$ to control the importance of the length in random walks in $\mu(k) = \lambda^k$ of Equation 4. Figure 3 shows that the classification accuracy ACC increases as $\lambda$ increases from 0.1 to 0.9, then it decreases after reaching the max value at $\lambda = 0.5$. Besides, 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 $\lambda$ is fixed to 0.5 in all the remaining experiments; with this setting, detection rates (TPR) reach 96.73%, with a false positive rate FPR of 0.73%.

Moreover, if we compare with the approach of [10] that uses standard API call graphs and standard random walk graph kernels to learn malware, we conclude that our extended API call graphs with our generalized random walk graph kernel give a better performance since by using API call graphs with the standard random walk graph kernel, we get a detection rate of 92.76% with 3.31%

false positive rates. The results are reported in Table 1 and Figure 4. Thus, our approach gives a better performance than the approach introduced in [10].

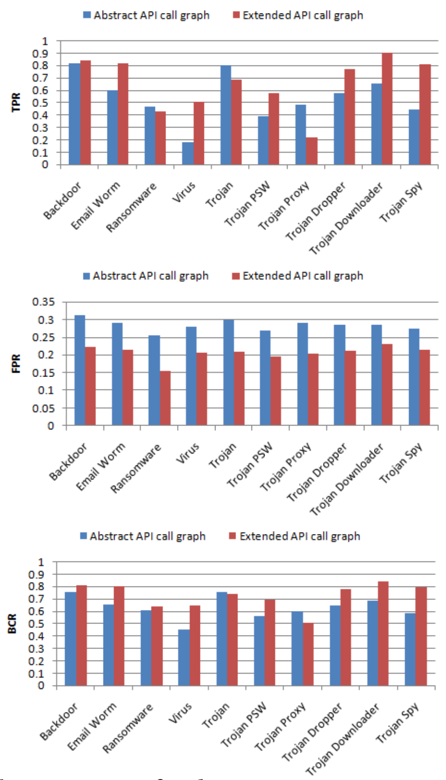

**Figure 5: Comparison of malware category recognition on our extended API call graphs vs. the abstract API call graph.**

**Performance of Malware Category Recognition.** Given an extended API call graph $g$ (with $h(g) \geq 0$), the goal is to assign it to one of 10 malware categories shown in Figure 2 based on $\arg\max_c h_c(G)$ (thanks to Equation 3). Figure 5 shows the class-wise TPR, FPR and BCR rates of our extended API call graphs with the generalized random walk graph kernel and their comparison against the results obtained using API call graphs with the standard random walk graph kernel. Figure 5 shows detection rate (TPR), false alarms (FPR) and balanced correctness rate (BCR) for each malware category. Our approach based on extended API call graphs and generalized random walk graph kernel led to a BCR of 72.52% while API call graphs with the standard random walk graph kernel led to 62.98%. Thus, our approach for malware category recognition based on extended API call graphs achieves a better performance than the approach in [10] that uses API call graphs.

## 6 CONCLUSION

In this work, we propose to use extended API call graphs to model binary programs. We implemented a generalized random walk graph kernel that computes similarities between extended API call graphs. Our experimental results show that our generalized graph kernel is able to capture well the structure of extended API call graphs since it leads to a detection rate of 97% with a low false alarm of 0.73%. These results outperform the ones obtained using API call graphs with the standard random walk graph kernel.

## REFERENCES

[1] VirusShare CryptoRansom 20160715. 2017. https://virusshare.com. (2017).
[2] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. 2011. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology* 7, 4 (2011).
[3] A. Bouajjani, J. Esparza, and O. Maler. 1997. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*.
[4] Christopher J. C. Burges. 1998. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Min. Knowl. Discov.* 2, 2 (June 1998).
[5] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011). Issue 3. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.
[6] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2007. Mining Specifications of Malicious Behavior *(ESEC-FSE '07)*. ACM.
[7] Vx Heaven Computer Virus Collection. 2014. http://vxheaven.org. (2014).
[8] K.H.T Dam and T. Touili. 2018. Precise Extraction of Malicious Behaviors. In *COMPSAC*.
[9] K. H. T. Dam and T. Touili. 2016. Automatic extraction of malicious behaviors. In *2016 11th International Conference on Malicious and Unwanted Software*.
[10] K. H. T. Dam and T. Touili. 2017. Malware Detection based on Graph Classification. In *ICISSP '17*.
[11] Erbiai Elhadi, Mohd Aizaini Maarof, and Bazara Barry. 2015. Improving the detection of malware behaviour using simplified data dependent api call graph. In *International Journal of Security and Its Applications*.
[12] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. 2010. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors *(SP)*.
[13] Dragos Gavrilut, Mihai Cimpoesu, Dan Anton, and Liviu Ciortuz. 2009. Malware detection using perceptrons and support vector machines. In *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. IEEE.
[14] Ban Mohammed Khammas, Alireza Monemi, Joseph Stephen Bassi, Ismahani Ismail, Sulaiman Mohd Nor, and Muhammad Nadzir Marsono. 2015. Feature Selection and Machine Learning Classification for Malware Detection. *Jurnal Teknologi* (2015).
[15] Joris Kinable and Orestis Kostakis. 2011. Malware Classification Based on Call Graph Clustering. *J. Comput. Virol.* 7, 4 (Nov. 2011).
[16] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. 2009. Effective and Efficient Malware Detection at the End Host.. In *USENIX security symposium*.
[17] Jeremy Z. Kolter and Marcus A. Maloof. 2004. Learning to Detect Malicious Executables in the Wild *(KDD '04)*.
[18] Deguang Kong and Guanhua Yan. 2013. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD*.
[19] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. 2003. On the Detection of Anomalous System Call Arguments. In *ESORICS*.
[20] Hugo Daniel Macedo and Tayssir Touili. 2013. Mining malware specifications through static reachability analysis. In *ESORICS*.
[21] Stavros D. Nikolopoulos and Iosif Polenakis. 2016. A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques* (2016).
[22] Chandrasekar Ravi and R Manoharan. 2012. Malware detection using windows api sequence and machine learning. In *International Journal of Computer Applications*.
[23] Internet Security Threat Report. 2018. https://www.symantec.com/security-center/threat-report. (2018).
[24] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. 2008. Learning and classification of malware behavior *(DIMVA '08)*.
[25] H. Sahbi and X. Li. 2010. Context-based support vector machines for interconnected image annotation. In *ACCV*.
[26] M.G. Schultz, E. Eskin, E. Zadok, and S.J. Stolfo. 2001. Data mining methods for detection of new malicious executables. In *S P 2001*.
[27] G. Tahan, L. Rokach, and Y. Shahar. 2012. Mal-ID: Automatic Malware Detection Using Common Segment Analysis and Meta-features. In *J. Mach. Learn. Res.*
[28] S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. 2010. Graph Kernels. In *J. Mach. Learn. Res.*
[29] Cynthia Wagner, Gerard Wagener, Radu State, and Thomas Engel. 2009. Malware analysis with graph kernels and support vector machines. In *MALWARE*. IEEE.
[30] L. Wang and H. Sahbi. 2013. Directed Acyclic Graph Kernels for Action Recognition. In *2013 IEEE International Conference on Computer Vision*.
[31] L. Wang and H. Sahbi. 2014. Bags-of-daglets for action recognition. In *ICIP*.
[32] Ming Xu, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren, and Ning Zheng. 2013. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques* 9, 1 (2013).