Exploring Graph-based Methods in Malware Detection

Yunhao Bai, Keith A. Johnson, Mengyuan Li
The Ohio State University

ABSTRACT

The rapid growth of smart devices and network scale has provide a solid base for malware development and spreading. As a result, detecting malware before it costs is crucial for the users of these smart devices such as laptops, desktops, etc. However, the virus detection schemes based on signature manifest unsatisfactory performance when they dispose the previously unknown virus. Recently, with the development of machine learning algorithms, we are able to predict a given software contains malicious component or not before it causes losses with the pre-trained classification schemes. In this report, we present a graph features based method, which can be used in the process of machine learning, and design a virus detection model based on our feature method. The features are extracted from Control Flow Graph (CFG) and Function Call Graph (FCG) of executable. we adapt a well-known three-step approaches in our detection model: 1) creating the corresponding graph, 2) processing the graph and extracting the useful features, and 3) training classifiers according to specific machine learning algorithms, and detect virus with the classifiers. Our evaluation is based on real malware and benignware collection, and shows an F1 score of 88.3% with the ten fold validation test.

1 INTRODUCTION

Malicious software, or malware, is any program that intends to halt or harm the users system. As a whole, malware has caused data and financial loss to individuals and companies to the tune of hundreds of billions of dollars [11]. Malicious software accomplishes this by stealing important identification information or holding it ransom. An example would be WannaCry, which infected hundreds of thousands of computers across over 150 countries worldwide [17]. WannaCry was able to make use of a vulnerability in Windows systems, called Eternal Blue, to access private data from computers. This makes malware a very real problem for computer scientists to tackle.

Malware as an industry continues to grow and adapt to current antivirus software, creating a need for new and more robust solutions to malware identification [11]. New vulnerabilities are discovered in systems as new systems are introduced, creating a game of cat and mouse between malicious hackers and antivirus software. Each step taken to increase the speed at which malware is identified reduces the cost malware inflicts on users and businesses.

Classic signature-based techniques rely on a large database of known malware to identify unknown programs [20]. These tables consist of previously identified malware and their string signatures, which are unique to byte sequences inside their binaries. Whenever an unknown program is scanned by antivirus software, the programs signature is extracted and checked against the table. If the signature matches any previous malware, the unknown program is flagged as malware. This results in a 100% detection rate for previously seen malware. However, this method cannot detect new malware. This method also cannot detect obfuscated malware when the obfuscation changes the string signature of a program. As a result, machine-learning methods are being developed and deployed to increase detection of zero-day malware.

The approach of this paper is to create a graph-based machine-learning model to detect zero-day malware. We use the semantic qualities of Control Flow Graphs (CFGs) extracted from programs to be able to identify malware that classic antivirus techniques cannot. Our hypothesis is that by comparing features from CFGs, we can train a classifier to successfully identify new unknown programs.

Lightspots. This paper has some lightspots to the study of graph-based malware detection in the following aspects:

- The paper explores the graph-based methods in Windows malware detection.
- The paper builds an automatic system to disassemble executable applications, extract features, train classifiers and verify unknown applications.
- The paper tries different classifiers and compare their performance with different measurements.
- The paper further give a proof-of-concept path-based method to identify malicious behaviors.

Roadmap. First, we will explain our decision to use CFGs and other relevant background knowledge in section 2. Then we will break down the design of our working model in section 3. Next, we'll go over the results from our experiments testing our model in section 4. In section 6, we'll talk about some possible future works. we'll include any relevant works from our peers as well as their inspiration into our paper in section 6. Finally, We'll share any conclusions we reached based on our results in section 7.

2 BACKGROUND

In this section, we present some background information of graph-based malware detection system.

Control Flow Graph (CFG). A CFG is a directed graph, where each vertex corresponds to a block of instructions of the original program [17]. Whenever a jump instruction appears in the code, an edge is created that points to where the jump lands. This results in a graph that describes the behavior of a program. An FCG is a simplified CFG where each vertex represents a function and each edge represents a function call. An FCG condenses information from a CFG, but also loses information on behavior inside functions.

Measures of the graph can then be extracted and compared against other CFGs, just as any other network graph. These measures can describe which instructions are the most used, or which instructions are never used. They can also describe the most common routes the program takes upon execution. As a result, these features can be used to distinguish malware from benign programs.

Classifiers. Machine-learning is a common way to improve antivirus detection of zero-day malware. These can range from feeding string signatures into a classifier to detecting malicious byte sequences [20]. However, graph-based methods are a more recent approach that leverages network science graph measures as features in a classifier These methods include using Heterogeneous Information Networks (HINs), Function Call Graphs (FCGs), and Control Flow Graphs (CFGs) [12] [20]. Recently, people pay more attention to automatically detect malware and some other famous works including [6–9, 13].

3 GRAPH-BASED MALWARE DETECTION SYSTEM DESIGN

The overall goal of our model is to be able to identify whether programs are benign or malicious given a set of training data. In classic antivirus techniques, new unknown programs are compared against the previous data to see if they match. Our approach is to train a classifier from extracted features of each programs' CFG.

As shown in Figure 1, building the model can be broken down into 3 steps: disassembling the training programs, extracting CFGs, extracting feature vectors, and training the classifier. After the classifier has been trained, it can be used to identify new unknown programs. Our system will then break down unknown program into its feature vectors and identify whether the program is a benign or a malware. In the next subsections under section 3, we'll go into detail how we accomplish each step.

3.1 Data Preparation

The first step is to extract graph from the training dataset and preprocessd the programs in the dataset, which includes disassemble and extract CFGs from the programs.

The dataset used by the model is a list of malicious and benign windows portable executable (.exe) files. The programs should come from a variety of different malicious and benign categories to improve performance. There should also be enough programs to train a linear classifier.

In order to disassemble and extract CFGs from the programs, the model uses IDA Pro [1]. IDA Pro handles disassembly and CFG extraction for most, if not all, Windows portable executable (.exe) files. We wrote a quick python script to let IDA Pro disassemble and extract CFGs from programs in batch, so that the model can handle datasets of 1000+ programs. We then wrote a python script to translate the resulting win32 (.gdl) files to edge list files. This way, we can feed the edge list files into networkx for quick feature extraction [16].

3.2 Feature Extraction

In order to use the CFGs as input for our classifier, we have to extract relevant features to use as feature vectors. From a glance, the graphs do not have any distinct features between malicious and benign programs. However, other research on Android malware found that centrality measures can distinguish malicious from benign apps [4]. Different types of malicious apps were also seen to have different densities in their CFGs [4]. To this end, we decided to extract eight handpicked features: average betweenness centrality, average closeness centrality, average degree centrality, average

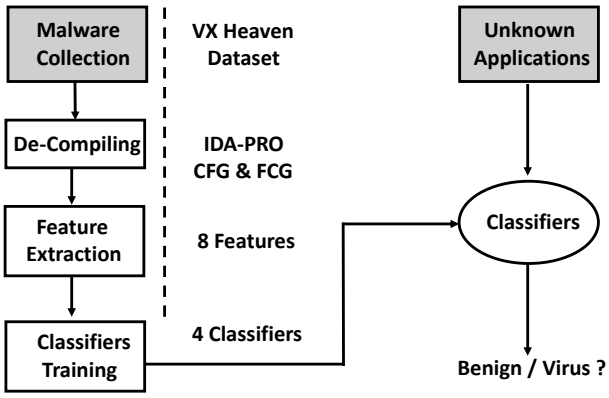


Figure 1: Workflow.

subgraph centrality, average clustering coefficient, average shortest path length, total number of nodes, and total number of edges. We do this by reading the graphs into networkx and using their built-in algorithms [16].

For each centrality feature, we take the average centrality measure over all nodes in the CFG. For betweenness centrality, we take the sum of the fraction of all-pairs shortest paths that pass through a node v. For closeness centrality, we take the reciprocal of the average shortest path distance to a node v over all $n-1$ reachable nodes. The average degree centrality is simply the average fraction of nodes each node is connected to, or the average normalized degree of the graph. For subgraph centrality, we take the sum of closed walks of all lengths starting and ending at a node u. Networkx accomplishes this by computing the eigenvalues and eigenvectors of the adjacency matrix. For the clustering coefficient, we take the fraction of possible triangles including a node u.

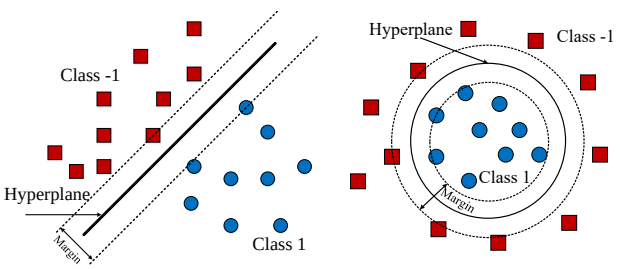
Unfortunately, the algorithms for betweenness and subgraph centrality are not feasible for some of the larger graphs. Therefore, we decided to prune nodes from those graphs until there were fewer than some threshold of edges before evaluating these measures. In the experiment, we pruned graphs until they had fewer than 5000 edges.

Once all the measures are calculated, the resulting feature vectors are then written into text files to be read by the classifier.

3.3 Classification

After we extract the eight features from the graphs, we can build the classifier based on the selected features. Here we try 10 different classifiers in total, and categorize them into four types: 1) Supported Vector Machine (SVM) based solutions, 2) K Nearest Neighbor (KNN) based solutions, 3) Decision Tree (DT) based solutions, and 4) other solutions.

3.3.1 SVM based solutions. SVM is a well-known classification scheme for binary classification problem. Simply stated, an SVM finds the best separating (maximal margin) hyperplane between the two classes of training samples in the feature space, as it is shown in Figure 2(a). Mathematically, to find the linear separator $\mathbf{w^T x + b}$ for the training samples, we need to solve a constrained quadratic optimization problem which minimizes $1/\|\|w\mathbf{w^T}^2$ plus



(a) Linear Separable **(b) Not Linear Separable**

Figure 2: An illustration of SVM classifier.

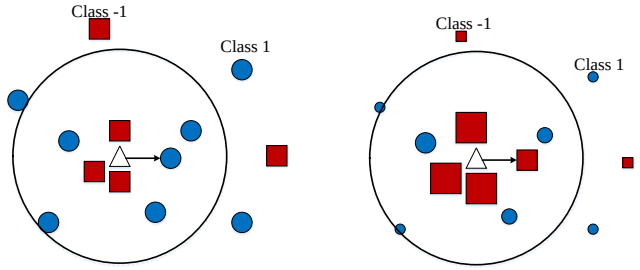
some penalty terms. After the training process, for a given executable files, we can extract its feature set x from the CFG, and map them in the feature space with $y = \mathbf{w}^{\mathbf{T}}\mathbf{x} + \mathbf{b}$. If $y > 0$, the given execuable is classified as a malware; Otherwise, it is predicted as a benignware. SVM is widely-used in many application such as activity recognition, emotion detection, etc [10][14].

However, not all sample set can be separated by a linear SVM. For example, in Figure 2(b), the separator is not a line but a circle. Foe this case, it is impossible to train a SVM classifier with a good performance. To solve this issue, some work has proposed to use kernel function to transform the samples to a higher dimension, where a linear support vector can be trained. For our scenario, since we cannot guarantee that the boundary between malware and benignware is linear, we also use two well-known kernel-based SVM for our scheme: Quadratic SVM and Gaussian kernel (RBF) SVM.

3.3.2 KNN based solutions. Besides SVM, KNN is another simple choice for the classification problem. Different than other machine learning schemes, KNN does not require any training process, and only need to store all the training samples in a database. When an unknown executable file comes and its features are extracted, we calculate the distance (usually Euclid) from this unknown sample to all the training samples. After that, the classification result is based on the majority voting based on types of K training samples that have the least distance to the unknown sample. Though simplicity, KNN can be proven to approach the performance of an optimal bayes classifier when K goes to infinity [15], and can be used in non-linear separation task such as shown in Figure 2(b).

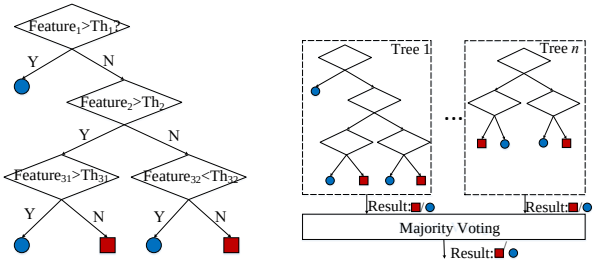
In addition to original KNN, here we also test the performance with weighted KNN. The difference between them lies in the voting process, where the weighted KNN shall assign a weight to each K nearest neighbor, which is often inversely proportional to the calculated distance. Figure 3 shows an example of difference between weighted and weighted KNN. as shown in Figure 3(a), the given sample shall be classified as a blue circle since there are four blue circle among its nearest neighbors. However, in Figure 3(b), the given sample shall be classified as a red square because those red squares in the training set are closer to the given sample. As a result, they shall be assigned a larger weight than those blue circles in the voting process.

3.3.3 DT based solutions. The decision tree is another choice of classification scheme. It recursively partitions the feature space to model the relationship between predictor variables and categorical



(a) Unweighed KNN **(b) weighed KNN**

Figure 3: An illustration of KNN classifier.



(a) A single decision tree. **(b) Multiple decision trees.**

Figure 4: An illustration of DT classifier.

response variable. Using a set of input-output samples a tree is constructed, and the popular learning algorithm for a decision tree includes ID3, C4.5 amd CART [19]. After that, pruning algorithms must be used to avoid overfitting issue. An exmaple of decision tree is shown in Figure 4(a), where several branch nodes and corresponding threshold values are trained. At each leaf node, the decision tree can yield the classification result.

However, a single decision tree may not be capable to do the classification task. Thus, in order to increase its robustness and performance, we also implement some ensemble learning based approaches to enhance the decision tree based solution. The idea of ensemble learning is shown in Figure 4(b), Several different decision trees are built, and each of them can output its own classification result. Then these result are combined with meta-decision maker that output the final classification result. The most known approaches of ensemble learning scheme includes boosting and random forest. Boosting shall train different decision trees with different weights in the training samples, and change these weights during the process. Random forest shall select different features randomly and build the tree for each selection. Generally, Using a random selection of features to split each node yields error rates that compare favorably to boosting algorithm, and are also more robust with respect to noise [19].

3.3.4 Other solutions. Besides these aforementioned classifiers, we also test some other classifiers such as logistic regression and Linear Discriminant Analysis (LDA). Logistic regression tries to model the likelihood for a class by using the sigmoid function, and define a loss function related to the binary classification problem. LDA is a method based on the projection, where it find the best projection plane for the two classes and get the separator on the projection

plane. The details for those two approaches are omitted due to the space limitation, and we direct interested reader to [19] for more details.

4 EVALUATION

In the evaluation section, we first introduce the experiment setup. We then test our schemes using the malware and benignware we collect using different classifier. At last, we show the improvement when combining the CFG with the Function Call Graph (FCG).

4.1 Experiment Setup

4.1.1 Platform Setup. We use the de-compiling tool IDA Pro to extract CFG from individual executable files [1]. We employ an IDC (IDC is IDA Pro's built in script language) program to create CFG automatically from disassembly, and the results are save by Win32 graph files. After that, we use another python script to convert all Win32 graph files to the edge list files that can be analyzed by NetworkX packages, and extract the features with NetworkX. We then use MATLAB R2018b to train the classifier and get the prediction result from the trained classifiers.

Table 1: Statistics of Executable Filesize in the Dataset.

Statistics	Filesize (malware)	Filesize (Benignware)
Maximum (KB)	15543	989
Minimum (KB)	2	1
Average (KB)	97	26
Median (KB)	28	17

4.1.2 Dataset Used. In our evaluation, we focus on the executable file for Windows called PE (Portable Executable). We use a well-known malware collection VX haven as our malware dataset [2], which consists of a total of 1532 PE files. Due to duplication and some compatible reasons, we can only get a total of 805 CFGs for the malware dataset. Most of the benignware were gathered from a freshly installed Windows 10 system files. Thus, we have a total number 1794 files consisting of 805 malware and 989 benignware. Table 1 displays the file size statistics for these two types of executables.

4.2 Correlation Analysis

Here we show the correlation for our selected features to evaluate whether they can be good feature set to train a robust classifier. We use the matrix of correlation coefficient to show the correlation between two features. The matrix size is 8×8 since we have 8 features in total, and each element a_{ij} shows the correlation between feature i and feature j in the range of $[0, 1]$. Generally, a lower correlation coefficient means that two features are not closed related to each other, and it is a indicator that the two selected features can provide different information about the CFG; Otherwise, the two features with large correlation coefficient can be seen as the same feature as they are closely related. Since some features are highly correlated, we do not choose Naive Bayes (NB) classifier in our scheme because NB classifier assumes all features are independent.

Figure 5 shows the correlation matrix for the selected eight features. We can see that the centrality metrics of the CFG are highly correlated such as closeness, betweenness, and degree centrality. The cluster coefficient shows little relation to other features, and it

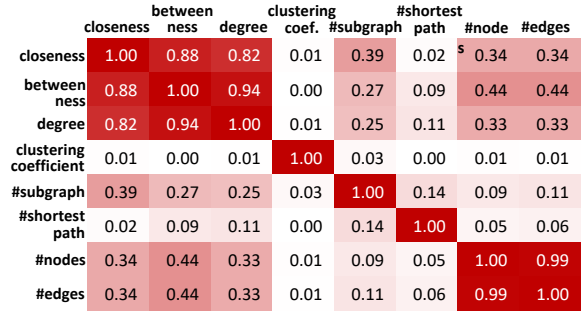
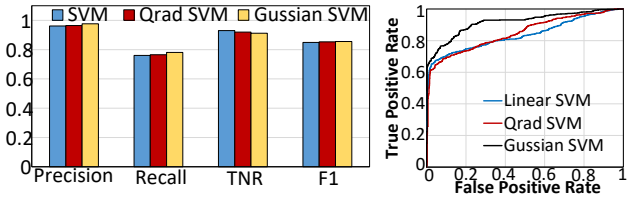


Figure 5: The correlation coefficient matrix for the eight selected features.



(a) F1 score **(b) ROC**

Figure 6: The performance of the SVM-based approaches.

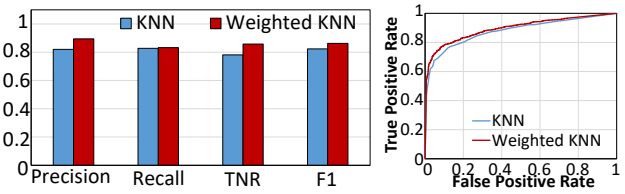
can be seen as a good feature to represent the CFG. The number of shortest paths and subgraphs also shows small correlation to other features, which is mainly caused by the sparse nature of the CFG compared with other graphs in the other cases. At last, the number of edges and nodes are highly correlated because more nodes indicate more edge generally.

4.3 Classification Result

In this part, we compare the performance of the selected classifiers in each categories in Section 3.C. We use the standard 10 fold cross-validation process in our experiments, and use the metric such as precision/recall, F1 score, and Receiver Operating characteristic Curve (ROC) to show the performance for each selected classifiers.

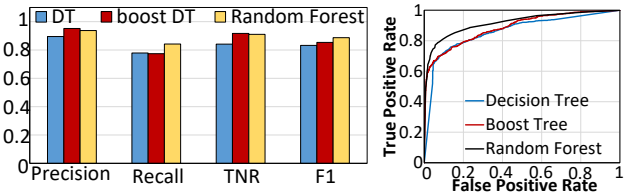
4.3.1 SVM performance. Figure 6 shows the prediction performance of SVM-based approach such as linear SVM, quadratic SVM and Gaussian SVM. They can all achieve an F1 score over 84.5%, with an accuracy of over 82%. We can see that the quadratic SVM slightly outperforms the linear SVM because the quadratic SVM allows non-linear separations between the two classes and introduce more flexibility for sample distribution. As a result, more robust boundary can be calculated . Moreover, the Gaussian SVM performance the best among all the SVM-based approaches. This is because it allow the samples in the original space to be project to a more higher dimension than that of a quadratic SVM, where a more accurate boundary can be found. Through these kernel-based SVM can improve the performance, the linear SVM can still achieve an F1 score of 84.9%, which is sufficient for malware detection.

4.3.2 KNN Performance. We then test the performance of KNN based solutions. Figure 7 shows the performance of the unweighted and weighted KNN classifier. We can see that the weighted KNN



(a) F1 score (b) ROC

Figure 7: The performance of the KNN-based approaches.



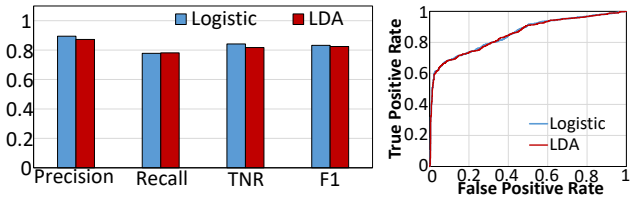
(a) F1 score (b) ROC

Figure 8: The performance of the decision tree based approaches.

outperform the unweighted version by 3.9% in terms of F1 score. The main reason of this phenomenon is that the werighted KNN consider the distance between the K nearest neighbors: the neighbor with a smaller distance should have a higher similarity to the given application than a farther one, and should be more decisive in the prediction. Though KNN classifiers performs worst than the SVM-based approaches, they still have several advantages: First, they are easy to build and do not require training. we only need to calculate the distance between the given sample and the reference samples in the database to do the prediction. In this way , the classifier can be easily adapted to a new scenario if we could know the result of our prediction. Second, KNN is robust to the outlier because it only consider the k nearest neighbor for classification. The outliers often have large distance to the valid samples and shall not be considered in the classification.

4.3.3 Decision Tree Performance. We then test the performance of decision tree based solutions. Figure 8 shows the performance of the selected decision tree based approaches, which can achieve an F1 score over 84%. Among all the approaches listed in Section 3.C, the random forest approach achieve the best performance with a F1 score of 88.3%. The main reason is that the ensemble learning approach leverage the combination of simple classifiers such as decision trees to work collaboratively, to improve the overall prediction performance. With multiple decision trees built upon different selection of features, the random forest solution can not only improve the performance, but also improve the robustness of the classification by reducing the over-fitting issue: Different decision tree can have different sensitivity of different features and the overall robustness can be improved over the single decision tree method (4.1% improvement over the single decision tree method in terms of F1 score in Figure 8).

4.3.4 Decision Tree Performance. At last, we test the performance of other linear separation approaches (i.e., logistic regression and LDA). Figure 9 shows the performance of the selected approaches,



(a) F1 score (b) ROC

Figure 9: The performance of other classification approaches (logistic regression and LDA).

which can achieve an F1 score over 80%. The logistic regression and LDA achieve an F1 score of 82.3% and 81.4%, respectively. The logistic regression performs slightly worse than the linear SVM because it does not consider the margin in this formulation. It only cares about the sigmod function of the likelihood for the two classes. The reason of the low performance of the LDA algorithm is still unknown to us and we will investigate it in the future work.

5 CFG-FCG COMBINATION: DISCUSSION AND FUTURE WORK

In this section, we show that it is possible to find some hidden malicious behaviors in malware by analyzing function call paths in FCG. We believe we can improve the performance by using both the features from the CFG and FCG and fix the classifier as the random forest for the comparison because it achieves the best performance among all the schemes.

5.1 Function Call Graph based Improvement

In this subsection, we will talk about our advanced approach to further imporve the detection accuracy by embedding features from Function Call Graph.

5.1.1 Intuition.

There are three main advantages of using FCG to check unknown applications' malicious behaviors. The first advantage is that malwares belonging to the same malware families will have similar FCG, especially for those malwares which are highly relied on function call from shared libraries. The second advantage is we can get more features from FCG. In CFG, we don't implement a semantic analysis towards different control blocks, which might miss some important information. However, functions in shared library will have exactly the same behaviors and thus by analyzing the FCG, we will be able to direct compare application's behaviors instead of analyzing graph features.

The third advantage is that some "smart" malwares may try to hide their malicious behaviors into a large scale of benign behaviors. The traditional malware detection system may not be able to detect those malwares for two reasons.

(1)The codes or blocks for malicious behaviors can be a small percentage code/blocks among the whole program. So, if the malware detection system lower their threshold, he may fail to detect the malware. (2) The malware can hide their function call paths by inject some benign but meaningless functions. For example, if a malicious behavior is achieved by $F_A \rightarrow F_B \rightarrow F_C$, the

malware can inject some other functions to hide this attempt like $F_A \rightarrow F_D \rightarrow F_B \rightarrow F_B \rightarrow F_F \rightarrow F_G \rightarrow F_C$.

Our approach will take advantages from these and further improve our detection system. We will then talk about the general steps for our FCG based improvement.

5.1.2 General Steps.

The general steps for FCG based improvement are shown below.

① **Collect Malware Families and Extract FCG.** The first step is to collect different malware families and use the IDA-Pro to extract the Function Call Graph. One key point here is to collect as many different malware families as possible.

② **Get Malicious Paths Pattern.** We first need to learn malicious function path pattern from different malware families. We will only focus on function calls from shared libraries. In order to do that, we will need to implement an additional process towards the FCG we get from the IDA-Pro to filter the malware's own functions. For example, when we try to process the function call path shown below:

$$F_{Shared_A} \rightarrow F_{Private_B} \rightarrow F_{Shared_C}$$

, where F_{Shared_A} and F_{Shared_C} are functions from shared libraries, $F_{Private_B}$ is the malware's own function (here we call it private function) and $\rightarrow$ denotes a existing function call.

So, after filtering those private functions, the remaining path we recorded would be:

$$F_{Shared_A} \rightarrow F_{Shared_C}$$

Note that it is very important to only gather malicious function path patterns. Some malwares may have some same benign behaviors and a manual check is needed here to make sure those common function call paths are actual malicious behaviors.

③ **Extract Features from Unknown Applications.** To detect unknown application, our system will first use IDA-Pro to extract the Function Call Graph. Same as the steps in ① and ②, we will only collect function calls from shared libraries. We also need to use the same filter in ② to filter malware's own private functions.

③ **Implement Pattern Percentage Algorithm.** The detection system will then implement the Pattern Percentage Algorithm to detect those malicious behaviors in the unknown application. The details of the algorithm is shown in Algorithm 1.

Our system will compare the unknown application's FCG with malicious pattern graph and calculate the similarity. The similarity is defined as the number of matched path divided by the total number of paths in the pattern graph. If the ratio is greater or equal to the threshold, we will predict this unknown application as a malware.

5.1.3 Limitations.

However, after extracting and analyzing FCGs in our dataset, we find we are not able to classify and find certain patterns for malware. There are mainly three reasons. (1) The dataset is not big enough and not classified as different malware families. Thus, we are not able to gather malwares with similar behaviors and find common function call paths. (2) This method is only valid in detecting those malwares which are highly relied on function calls to implement different functionalities. In our dataset, many malwares are small

Algorithm 1: Pattern Percentage Algorithm to Detect the Unknown Applications.

```
initialization;
for Each start function F_0(in pattern graph P_0) in the malicious function path
  set do
    int matched_path = 0; int Num_path = number of paths in P_0;
    if F_0 is inside the FCG (F_0 is called by the app) then
      for each path (the start node and the end node are F_m and F_n) in P_0
        do
          if FCG contains F_m and F_n  path between F_m and F_n are
            matched then
            matched_path = matched_path + 1;
      Calculate the similarity using matched_path/Num_path
      if matched_path/Num_path >= Threshold then
        Predict as a malware;
        Break;
      else
        Continue;
    else
      Continue;
    Predict as a benign;
```

malwares (e.g., the size is small), which means they may not use certain function calls to conduct something malicious. Our dataset is a Windows dataset. The FCG methods should work well in Android applications since it is most common in detecting malwares in Android applications since Android applications are highly relied on function calls to implement different functionalities.(3)The malwares in our dataset don't try to hide their malicious behaviors into a benign. Our approach targets at some "smart" malwares which will try to hide their malicious behaviors into a large scale of benign behaviors. However, because of the size of dataset is not larger enough and most malwares are straight forward and old, so the dataset is not compatible for our approach.

5.2 Future Work: Dynamic Execution Graph

Dynamic Control Flow Graph (DCFG) is an extension of existing CFG [5]. A sample example of how DCFG works is shown in Figure 10. Unlike traditional CFGs which focus on all blocks, DCFG monitor applications dynamically and will only focuses on those blocks which have been executred. At the same time, it will also records the number of times each edge and block are executed. A very famous tool used in industry is called Pinplay from Intel [18].

We believe the performance will be much higher if we try to implement a DCFG-based malware detection system. All redundant and confusing blocks will be removed automatically and we will be able to grab more features since those DCFG tools will also records the number of times each edge and block are executed.

However, many extra efforts are needed here. A dynamic analysis means we need to actual execute those applications and malwares. Some malwares may crash the OS immediately so how to carefully handle it is a problem. Meanwhile, it is hard to use script to automatically generate DCFG in Windows.

6 RELATED WORKS

6.1 Android Malware Detection

Malware detection for Android applications can be a very similar problem. Android malware is just as prevalent as windows malware,

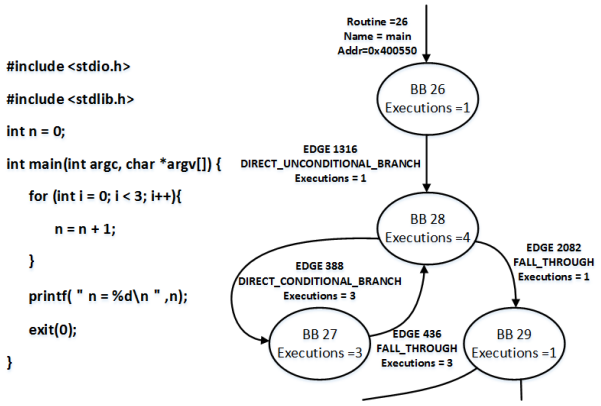


Figure 10: Dynamic Control Flow Graph: An example. (This figure is copied from a paper Mengyuan cooperated before).

if not more so. It is also just as lethal. Although our paper addresses windows programs, it can be applied to Android applications if the training data were Android applications, as IDA Pro can disassemble Android applications and extract their CFGs. On the other hand, many papers find new ways to detect malware specifically for Android applications.

HinDroid. In [12], they create features for their classifier by taking lists of API calls and constructing a heterogeneous information network (HIN). A way to detect semantic similarities between Android applications is to look at their API calls. Android applications will make API calls whenever they are asking or invoking permissions granted by the user. By constructing an HIN from the API calls, the authors can use the resulting meta-paths as features for their multi-kernel classifier. From this paper, we decided to try non-linear classifiers for our own model.

DroidNative. In [3], the authors attempt to look inside the native code of Android applications for malware. Sometimes, Android applications have parts of their programs in native code to either use legacy code or address hardware dependencies. Thus, they combine native code and byte code into one disassembled code. The authors proved in their results that adding native code to their detection model increased performance. The authors do admit that their model has over 14,000 lines of code. This was a factor in our decision to focus on windows programs, or at least to not consider native code in our model. Our approach was to try and find a combination of features and classifier that gave the best results, which could be accomplished on windows programs.

IOT Malwares. In [4], the authors analyzed Android malware to try and find distinctions between Internet of Things (IoT) malware. They find that IoT malware is much denser than Android malware. What's more important to our paper is that they find that centrality measures are an effective way of finding distinctions between IoT malicious and benign programs. They prove this by building a classifier using betweenness centrality, closeness centrality, degree centrality, shortest path, density, total number of edges, and total number of nodes. Our model closely resembles this classifier in terms of feature extraction [4].

6.2 Windows Malware Detection

Some other malware detection techniques for windows programs influenced our paper. In [17], the authors used lazy-binding CFGs as features for their classifier. They also applied deep learning to their classifier. Their model made use of IDA Pro, which is common for models targeting windows programs. They also included malware from VXHeaven, among other sources, which lead to our decision to use VXHeaven as training data. While their intention was to defend against complex malware like WannaCry, they also proved that using more complex classifiers improves performance, which helped our decision to explore different classifiers.

In [20], the authors use a shortlist of features from each programs function call graph (FCG) as features. They included features related to function types and subgraph measures. They also use VXHeaven in their dataset. This influenced our decision to include the subgraph centrality as a feature and VXHeaven as a dataset for our experiments.

7 CONCLUSION

The recent growth of machine learning algorithm has introduced new ways for virus detection and prediction. In this paper, we have proposed a malware detection scheme based on the Control Flow Graph (CFG) and Function Call Graph (FCG) and machine learning algorithm. Our design features a three-step approach to detect malware beforehand: 1) graph extraction, 2) feature extraction from the graphs, and 3) classification. Our evaluation results shows that our scheme can achieve an F1 score of 88.2% with random forest classifier.

REFERENCES

[1] 2020. IDA Pro Disassembler. (2020). https://www.hex-rays.com/products/ida/
[2] 2020. VX Heavens Virus Collection. (2020). https://archive.org/details/vxheaven-windows-virus-collection
[3] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2016. Droidnative: Semantic-based detection of android native code malware. *arXiv preprint arXiv:1602.04693* (2016).
[4] Hisham Alasmary, Aminollah Khormali, Afsah Anwar, Jeman Park, Jinchun Choi, DaeHun Nyang, and Aziz Mohaisen. 2019. Poster: Analyzing, Comparing, and Detecting Emerging Malware: A Graph-based Approach. (2019).
[5] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.
[6] 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), 247–258.
[7] Duen Horng "Polo" Chau, Carey Nachenberg, Jeffrey Wilhelm, Adam Wright, and Christos Faloutsos. 2011. Polonium: Tera-scale graph mining and inference for malware detection. In *Proceedings of the 2011 SIAM International Conference on Data Mining*. SIAM, 131–142.
[8] Ammar AE Elhadi, Mohd A Maarof, and Ahmed H Osman. 2012. Malware detection based on hybrid signature behaviour application programming interface call graph. *American Journal of Applied Sciences* 9, 3 (2012), 283.
[9] Ammar Ahmed E Elhadi, Mohd Aizaini Maarof, and BI Barry. 2013. Improving the detection of malware behaviour using simplified data dependent api call graph. *International Journal of Security and Its Applications* 7, 5 (2013), 29–42.
[10] Salvatore Gaglio, Giuseppe Lo Re, and Marco Morana. 2014. Human activity recognition process using 3-D posture data. *IEEE Transactions on Human-Machine Systems* 45, 5 (2014), 586–597.
[11] John F Gantz, Alejandro Florean, Richard Lee, Victor Lim, Biplab Sikdar, Sravana Kumar Sristi Lakshmi, Logesh Madhavan, and Mangalam Nagappan. 2014. The link between pirated software and cybersecurity breaches. *IDC White Paper* (2014).
[12] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. 2017. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1507–1515.

[13] Keehyung Kim and Byung-Ro Moon. 2010. Malware detection based on dependency graph using hybrid genetic algorithm. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. 1211–1218.
[14] Yi-Lin Lin and Gang Wei. 2005. Speech emotion recognition based on HMM and SVM. In *2005 international conference on machine learning and cybernetics*, Vol. 8. IEEE, 4898–4901.
[15] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.
[16] NetworkX developer team. 2014. NetworkX. (2014). https://networkx.github.io/
[17] Minh Hai Nguyen, Dung Le Nguyen, Xuan Mao Nguyen, and Tho Thanh Quan. 2018. Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning. *Computers & Security* 76 (2018), 128–155.
[18] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 2–11.
[19] Stuart Russell and Peter Norvig. 2002. Artificial intelligence: a modern approach. (2002).
[20] Zongqu Zhao, Junfeng Wang, and Chonggang Wang. 2013. An unknown malware detection scheme based on the features of graph. *Security and Communication Networks* 6, 2 (2013), 239–246.