Hybroid: Toward Android Malware Detection and Categorization with Program Code and Network Traffic

Mohammad Reza Norouzian[1⋆], Peng Xu[1⋆], Claudia Eckert[1], and Apostolis Zarras[2]

[1] Technical University of Munich
[2] Delft University of Technology

Abstract. Android malicious applications have become so sophisticated that they can bypass endpoint protection measures. Therefore, it is safe to admit that traditional anti-malware techniques have become cumbersome, thereby raising the need to develop efficient ways to detect Android malware. In this paper, we present *Hybroid*, a hybrid Android malware detection and categorization solution that utilizes program code structures as static behavioral features and network traffic as dynamic behavioral features for detection (binary classification) and categorization (multi-label classification). For static analysis, we introduce a natural-language-processing-inspired technique based on function call graph embeddings and design a graph-neural-network-based approach to convert the whole graph structure of an Android app to a vector. For dynamic analysis, we extract network flow features from the raw network traffic by capturing each application's network flow. Finally, *Hybroid* utilizes the network flow features combined with the graphs' vectors to detect and categorize the malware. Our solution demonstrates 97.0% accuracy on average for malware detection and 94.0% accuracy for malware categorization. Also, we report remarkable results in different performance metrics such as F1-score, precision, recall, and AUC.

1 Introduction

Android has become the most popular mobile operating system worldwide. Unfortunately, it has become a primary target platform for attackers using Android to launch millions of malicious applications due to its prominence. Attackers dupe victims to reveal their sensitive information or perform malicious operations, such as spying on users, propagating spam, or launching unwanted advertisements. Simultaneously, investigation of Android malware, which includes malware detection and categorization, has become crucial for security researchers and experts in both academia and industry. As a result, numerous research studies have attempted to detect and categorize Android malware [5, 10, 17, 21, 27–29, 32].

⋆ These authors have contributed equally to this work and share first authorship

Numerous approaches leverage the contextual information of Android applications, yet nearly none of them can monitor malware behaviors if we use contextual information statically. For example, Li et al. [14] presented a classifier using the Factorization Machine architecture, which extracts various Android application features from manifest files (e.g., permissions and intents) and source code (API calls). Similarly, Chen et al. [6] proposed an approach that detects Android malware with Android application's static features, such as permissions, components, and sensitive API calls. Although these methods add an extra security level to the Android platform, they come with their limitations, particularly for those obfuscated applications when executed [7]. This problem can be mitigated by introducing dynamic analysis, which monitors malware actions and analyzes the captured behavior when running in a sandboxed environment.

In essence, similar to static analysis, there are two types of dynamic analysis target Android applications. The first focuses on system-level behavior, extracting features from API usage or system calls, whereas the latter extracts features from network-level actions (i.e., data received or sent over the network). Analyzing system-level malware behavior is expensive and slows down the processing speed. In contrast, analyzing network-level activities is scalable and more cost-efficient, while it often exposes the core behavior of malware when trying to communicate with the attacker. Specifically, it can reveal the exfiltrated information and the commands sent or received by the malware. From a network perspective, monitoring and analyzing a system that extracts behavioral information from the user causes less overhead on the end hosts. To detect legitimate and malicious behavior, several approaches utilize the network traffic pattern of Android applications [2, 15, 16, 25, 32]. Most of them concentrate on the manual indicated features and build rule-based classifiers for detecting Android malware. Sadly, sophisticated attacks can easily evade network-rule-based methods since rule-based analysis relies on distinguishing expected versus anomalous behavior; these methods may suffer when malware is modified to hide its footprints or behavior. However, one of the main challenges of analyzing network-level activities is related to their offline inspection behavior.

In this paper, we present *Hybroid*, a hybrid framework for Android malware detection and categorization based on static and dynamic features to overcome the drawbacks mentioned above. From the users' point of view, *Hybroid* does not change anything of the Android application itself. We take the program code inside apps as input for static analysis and present a *Natural Language Processing (NLP)* inspired method based on the function call graph, which detects obfuscated applications. In brief, we first design the `opcode2vec`, `function2vec`, and `graph2vec` components to represent instructions, functions, and the entire program's information with vectors. Next, we take network traffic as input and extract 13 features for dynamic analysis. Finally, we combine static and dynamic analysis features and feed them into the machine learning and deep learning networks for training and prediction. Our results show that *Hybroid* outperforms most existing frameworks, as we get 97.0% accuracy for malware detection and 94.0% accuracy for malware categorization on average.

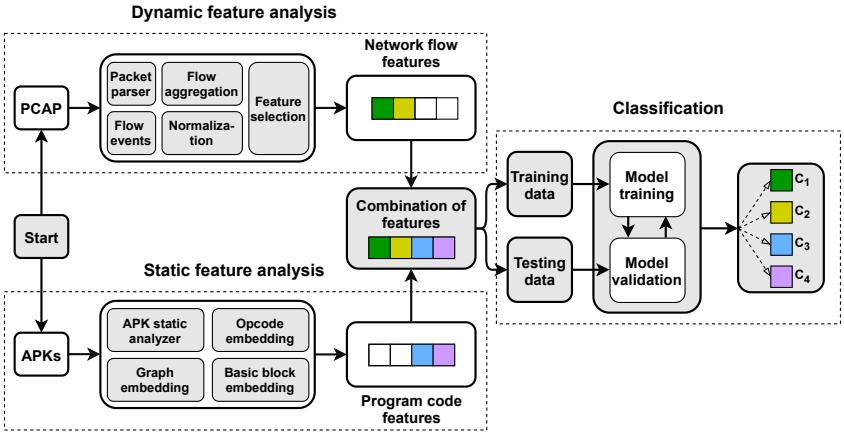

Fig. 1. *Hybroid* architecture

In summary, we make the following primary contributions:

- We present and open source *Hybroid*,[1] a hybrid framework for Android malware detection and categorization based on static and dynamic features.
- We design and implement automatic extraction of flow-based features from the Android raw network traffic as a dynamic feature.
- We leverage NLP and convert machine codes, functions, and programs to opcode2vec, function2vec, and graph2vec by embedding methods.
- We evaluate the accuracy of our approach using a real-world dataset and show that *Hybroid* outperforms nearly all state-of-the-art solutions.

2 System Design

In this section, we describe the architecture of *Hybroid* (see Figure 1), which comprises static and dynamic features. We extract static features by studying the *Control Flow Graph (CFG)* of the Android bytecode and the dynamic features by investigating the network flow data. Next, we combine these two groups of features as input vectors to train a machine learning model. Essentially, our approach is divided into three main parts: static features preparation (features from program code), dynamic features preparation (features from network traffic), and machine learning classification.

2.1 Static Features Preparation

Before getting into our methodology's details, we have to extract the opcode, basic block, and CFG from the Android APKs (Android application package).

[1] https://github.com/PegX/Hybroid

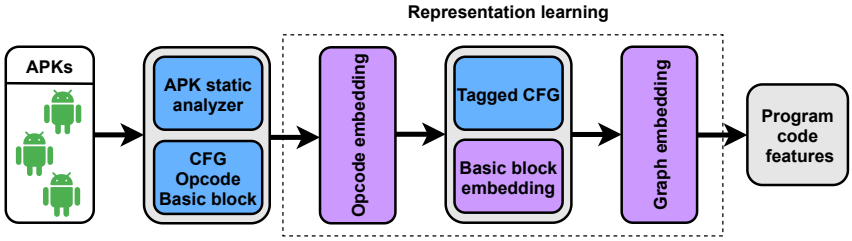

Fig. 2. Converting program code to vector

We extract the CFG by utilizing the Androguard framework (APK static analyzer) and iterate each function in the program to get the basic block for each function (method).[2] Furthermore, we analyze each instruction and take opcode as our basic term. After obtaining the opcode, basic block, and CFG, our primary approach is presented as follows. For the packing and obfuscated apps, similar to Xu et al. work [26], Androguard can also help our *Hybroid* to extract CFG and opcode from the apps, and we can also construct our graph structure.

Figure 2 depicts an overview of the steps involved in extracting features from the code graph structure. The entire process includes three main steps: (*i*) opcode embedding that converts the machine instructions into vectors, (*ii*) basic block embedding that transforms a basic block of the program into a vector (basic blocking embedding is done with Tagged CFG, which is used to combine multi-opcode to a vector), and (*iii*) graph embedding that modifies the whole function call graph into a vector. Finally, during the conversion of the opcode, basic block, and function call graph into vectors, we utilize representation learning techniques to learn the essential model parameters for getting the final 64-bit vector.

Representation Learning. To generate the node attribute in the CFG, we leverage representation learning. Representation learning [4], which can learn features from raw data automatically, has increasingly attracted researchers' and engineers' focus. Compared to those manually indicated attributed control flow graph (ACFG) methods, like Xu et al. [30], Adagio [10], and Yan et al. [31], *Hybroid* can extract ACFG automatically without preparing manual features and avoiding the challenge of manual indicated methods (how to pick up the useful features is a challenge) because of the representation learning. Additionally, *Hybroid* borrows ideas from Natural Language Processing to assist the feature engineering. It uses the word2vec to convert instructions to vectors and automatically learns the vector from the basic block's raw instruction.

In brief, *Hybroid* static analysis part introduces representation learning as the fundamental technique to represent code and use the control flow graph as fundamental to organize the program. Additionally, it utilizes NLP to convert the byte sequences (instruction and basic block) to vectors, used to replace the manually indicated features [10, 31]. *Hybroid* then feeds those generated

[2] `https://github.com/androguard`

vectors into a learning-based classifier to extract static features. In other words, *Hybroid* uses the transform learning technique to use the previously trained instruction2vec model to convert the byte sequences to vectors.

Opcode Embedding. To simplify the procedure, we replace instruction (opcode and operands) embedding with opcode embedding. The reason for this replacement is the following. First, the opcode represents Dalvik's instruction behaviors, whereas the operands represent the parameters. Dalvik's operands are virtual registers in a virtual machine. Those values are significantly affected by the underlying usage of Dalvik VM or ART VM. Thus, it is not possible to enumerate them all. Additionally, if various malware samples in the same family use the same malicious pattern, the opcode itself can capture these behaviors.

In theory, our opcode embedding method may suffer from the *operand removal* problem [11]. A significant issue with operand removal is that all the *Invoke-Virtual* instructions have the same embedding vector, no matter what are the targets of the *Invoke-Virtual* instructions.[3] For the opcode embedding method, or `opcode2vec`, we map each opcode $\text{op}_i \in \text{OP}$ (where OP stands for the whole Dalvik opcodes) to a vector of the real number, using the `word2vec` model with the skip-gram method [18]. `word2vec` is an excellent feature learning technique, which is based on continuous bag-of-word and skip-gram techniques. The skip-gram learning technique uses the current opcode to predict the surrounding opcodes. We train our `opcode2vec` model with a large corpus of opcodes extracted from real applications.

Basic Block Embedding. In this work, we treat the basic block embedding in the control flow graph similarly to the sentence embedding in the natural language processing. Overall, we introduce our method for performing the basic block(nodes in control flow graph) embedding, which are described as follows. We utilize the weighted mean of a non-empty finite multi-set of instruction's opcode to calculate the basic block embedding. Assuming the function f includes n-opcode and a l-dimensional vector represents each opcode, the weight of the corresponding non-negative weights $w_1, w_2, \ldots, w_n$ are given as: $\vec{\tilde{f}} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$, where x_i represents the l-dimensional opcode embedding and w_i stands for the weighted of each opcode.

Graph Embedding. After deriving the basic block embedding, we take the generated basic block embedding as the node embedding of the control flow graph. In other words, we perform graph embedding on a control flow graph level. The module's ultimate purpose is to convert the graph representation into a vector and then feed it as input for the neural network-based classifier. We take structure2vec [9] graph embedding method to convert one graph to a vector.

We utilize the Equations (1), (2), and (3) to convert a control flow graph to a graph-vector, which stands for the whole Android application. In our work, our graph-based control flow graph embedding includes two components. The first

[3] All the calling instructions such as *invoke-super*, *invoke-direct*, *invoke-static*, and *invoke-interface* suffer from the same problem.

Algorithm 1: Graph embedding

Input: Instruction embedding $v_i : i \in I$, control flow graph insider of a function g_f, parameter α

Output: Graph embedding $v_f : f \in F$

Initialize $\mu_v^0 = \vec{Rand}, forall v \in V$
for *t=1 to T* **do**
  for $v \in V$ **do**
    $l_v = \sum_{u \in N(v)} \mu_u^{(t-1)}$
    $\mu_v^{(t)} = tanh(W_1 x_v + \sigma(l_v))$
$v_f = W_2(\sum_{v \in V} \mu_v^T)/len(V))$
return v_f

one is the control flow graph extraction, and the other is the graph embedding for each control flow graph, which is adapted from the structure2vec.

The graph vectors (nodes) are basic blocks for graph embedding, and the edges are connections among those basic blocks in the CFG. Each vector (node) contains a set of opcodes inside it. The basic block embedding constructs each node's feature. Finally, a p-dimensional vector μ_i is associated with each vertex v_i. We use adapted `structure2vec` to dynamically update the p-dimensional vector μ_i^{t+1} during the training of the network. The updating process is executed as follows:

$$\mu_v^{(t+1)} = F(x_v, \sum_{u \in N_v} \mu_u^{(t)}), \forall v \in V. \tag{1}$$

We randomly initialize the $\mu_v^{(0)}$ at each vertex. In practice, we design the function F as follows:

$$F(x_v, \sum_{u \in N_v} \mu_u^{(t)}) = tanh(W_1 x_v + \sigma(\sum_{u \in N(v)} \mu_u)) \tag{2}$$

For an effective nonlinear transformation $\sigma(.)$, we define $\sigma(.)$ itself as an n layer fully-connected neural network and the W_1 is trainable parameter.

$$\sigma(l) = P_1 * ReLU(P_2 * \ldots ReLU(P_n l)) \tag{3}$$

The overall CFG-based embedding algorithm is illustrated in Algorithm 1. The graph embedding generates the vector embedding after all iterations, and we use the average aggregation function as our last step to transform the vector embedding to the graph-based function embedding.

After deriving our graph embedding for the function call graph, we design a two-layer MLP (multilayer perceptron) network as our representation learning network to learn parameters used to convert the program code into vectors. In our network, malware detection is a binary classification issue. We label malware samples as `1` and benign samples as `-1` at training. During testing, we treat all predictions less than zero as benign and the rest as malicious.

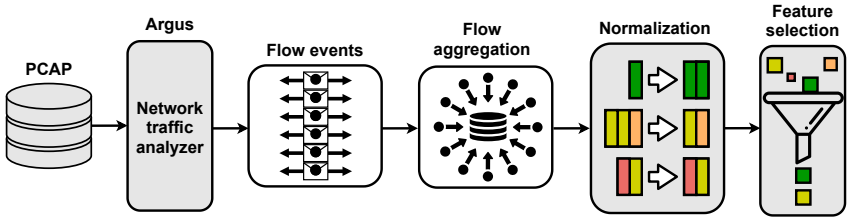

Fig. 3. Dynamic analysis overview

$$v_{f(G_h)} = \alpha * ((< g_i, w_{i1} > + b_{i1}), w_{i2} + b_{i2}) \tag{4}$$

where the $w_{i1}, w_{i2} \in R^p$ is the weight of the two-layer MLP network and the $b_{i1}, b_{i2} \in R^p$ is the offset from the origin of the vector space. In this setting, a function call graph G_h is classified as malicious if $f(G_h) > 0$ and benign if $f(G_h) < 0$. The vector $v_{f(G_h)}$ that corresponds to $f(G_h)$ is collected as the final representation of the program code. By using the above-stated methods, we finally get a 64-bit vector representing the whole program code and present the static features of program code.

Last but not least, we should mention the transductive and inductive embedding. Our work relies on word2vec to convert instructions to vectors. This requires relying on a large and representative dataset to train the embedding: word2vec is a transductive approach and requires access to the entire alphabet. As our method focuses on instruction mnemonics, our transductive approach of word2vec did not influence the final results since graph embedding (convert the control flow graph to vectors) is an inductive approach in which graphs of the testing dataset are unknown at training time.

2.2 Dynamic Features Preparation

Figure 3 illustrates an overview of dynamic analysis (i.e., extracting features from network traffic). The whole process involves three main steps. The first step is the network flow generation that involves converting raw network traffic into network flow events. Alternatively, we could use deep packet inspection to extract network traffic features to understand the malware behavior better. However, this tactic cannot be applied to most real-world scenarios due to privacy concerns. In contrast, high-level flow features do not necessarily render a correct picture of malware behavior. To address this gap, we leverage static analysis, combining it with dynamic network analysis. In the second step, we normalize the flow features extracted from Argus,[4] and in the last step, we use feature selection mechanisms to reduce and finalize our dynamic feature set.

Network Flow Generation. The raw data (PCAP files) captured from each application network traffic is fed into a packet parser to analyze network behav-

[4] `https://openargus.org/`

iors. Our proposed solution uses the Argus network traffic analyzer to handle the first phase of our dynamic analysis. Argus is an open-source tool that generates bidirectional network flow data with detailed statistics for each flow. Argus defines a flow by a sequence of packets with same values for five tuples that are *Source IP*, *Source Port*, *Destination IP*, *Destination Port*, and *Protocol*.

However, the output values of Argus features' are almost numeric, except for two categorical values: direction and flag states. To map them into discrete values, we use the one-hot encoding that encodes categorical features as a one-hot numeric array for our feature generation. The output of Argus involves numerous flow events with around 40 feature sets related to each flow.

Since there are at least more than one flow events for each PCAP file, the next step is to map each bunch of flow events into one data sample. To handle this step, we aggregate the values of network flow features by calculating the mean values, appending them to a single record. These steps mentioned above perform as a preprocessing phase, which converts the raw network data into numeric values that create a dataset ready to train any machine learning model.

Normalization. The extracted features must be normalized before being given to the classification algorithms since their values vary significantly. For example, if we chose Euclidean distance as a distance measure for classification, normalization guarantees that every feature contributes proportionally to the final distance. To achieve normalization, we use min-max scaling as shown below:

$$x^1 = (x - min(x))/(max(x) - min(x)) \tag{5}$$

where $min(x)$ and $max(x)$ represent range values. This method returns feature values within the range [0,1]. An alternative method would be using standard scaling by subtracting the mean values of the features and then scaling them to unit variance. However, this method would mitigate the differences in the values, making the detection harder (we examined this experimentally).

Feature Selection. Selecting features is critical, as it affects the performance of the model. There exist two main reasons to reduce the number of features:

1. *Complexity Reduction:* When the number of features increases, most machine learning algorithms require more computing resources and time for execution. Thus, reducing the number of features is essential for saving time and resources.
2. *Noise Reduction:* Extra features do not always help to improve the algorithm performance. In contrast, they may produce severe problems related to model overfitting. Therefore, selecting a set of useful features reduces the possibility of model overfitting.

Before the training and testing phase, we implemented various feature selection algorithms to find the best set of final features for our analysis. We used three feature selection algorithms: Pearson Correlation, Extra Trees Classifier (extremely randomized trees), and a Univariate feature selection (select features according to the highest k scores). At the end of the process, we selected 13

Table 1. List of network flow features

Notation	Traffic Features
Mean	*Average duration of aggregated records*
sTos	*Source TOS byte value*
dTos	*Destination TOS byte value*
sTtl	*Source to destination TTL value*
dTtl	*Destination to source TTL value*
TotBytes	*Total transaction bytes*
SrcBytes	*Source to destination transaction bytes*
DstWin	*Destination TCP window advertisement*
SrcTCPBase	*Source TCP base sequence number*
DstTCPBase	*Destination TCP base sequence number*
Flgs_er	*State flag for Src loss/retransmissions*
Flgs_es	*State flag for Dst packets out of order*
Dir	*Direction of transaction*

network flow features for our final dynamic analysis feature engineering. These selected features describe the general behavior of the network activity for each data sample and can be found in Table 1.

However, we perform an extra analysis to explore the quality of selected features that are highly related to the target labels. We assume that any two features are independent without being redundant. To investigate the redundancy score, we use Kendall's correlation method (Figure 4). Notice that any two independent features are interpreted as redundant if the correlation score is extremely high, whereas a high correlation between dependent features is desired.

Observation of Malware Network Communications. We check the type of communication to spot if the applications use secure communication channels or transmit the data on unencrypted flows. We can make observations about the entire encrypted data flows instead of just the handshake or individual packets. This is done by extracting the features of each data record by flow-level instead of packet-level approach. As we can see in Table 2, a relatively small number of applications are using encryption for communication. When we compared malicious to benign applications traffic, we found out that the communications that initially start with more upload than download traffic are more likely to be malicious. The reason is that when malware connects to a control server, it often identifies itself with a client certificate, which is rarely seen during normal TLS usage. Another aspect we notice is that after the initial connection to the control server has been established, the channel is often kept open but idle, with only regular keep-alive packets being sent.

When comparing these two aspects with what ordinary TLS traffic created in an HTTPS session in a browser looks like, one can easily see a very different behavior: when requesting a website, the initial upload usually consists only of a GET request (little upload), with a large response in the form of web page content being sent from the server (large download). However, *Hybroid* results

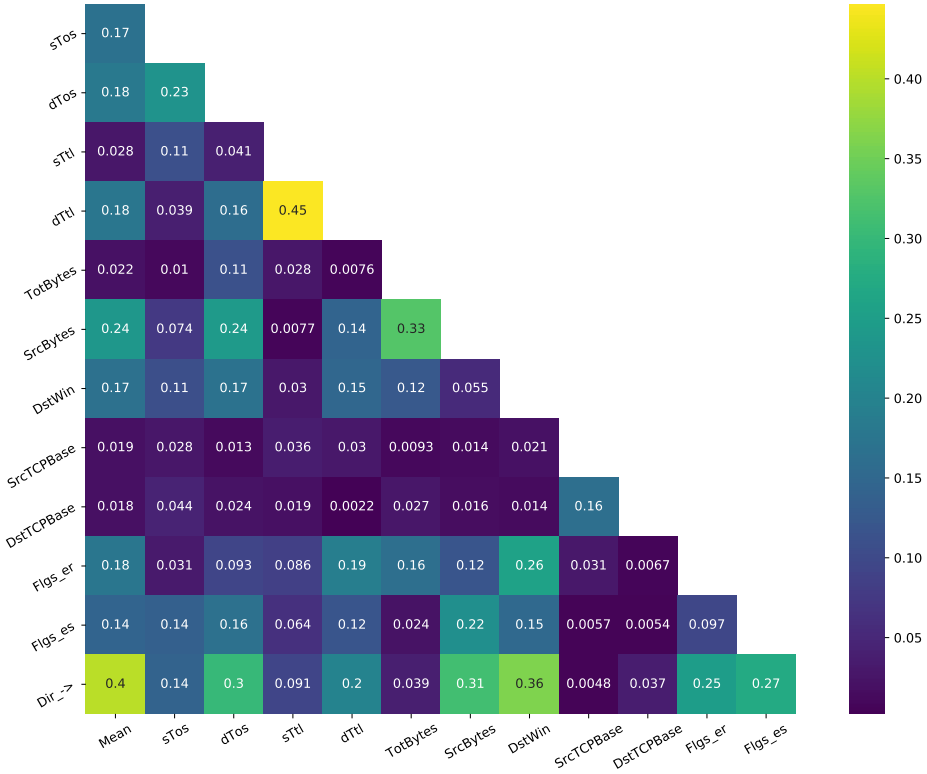

Fig. 4. Dynamic network flow feature correlation scores

(see Section 3.5) of malware detection and categorization show that analyzing flow metadata would be effective on encrypted flows too.

2.3 Machine Learning Classification

The classification aims to detect and categorize the APK samples, whether the source APK is a benign application or a specific type of malware. In the beginning, we tested various supervised learning algorithms (support vector machines, naive Bayes, decision tree, random forests, and gradient boosting) to assess classifier performance. The differences of using various learning algorithms confirm our methodology that the selected static and dynamic features help to identify the distinction between benign and malicious APKs. Test results revealed that SVM and naive Bayes demonstrated the worst performance and were thus excluded from the tests.

For model validation, we used the cross-validation technique to test whether the model can predict new samples that were not used in previous estimations. The intention for cross-validation is to reduce the chance of overfitting or selection bias and improve the model's generalization to an independent dataset.

Table 2. Type of malware category communication networks

Category	HTTP Flow	TLS Flow
Adware	52.00%	8.00%
Ransomware	29.22%	0.00%
Scareware	61.38%	10.89%
SMSmalware	52.20%	10.28%

3 Evaluation

We use different types of machine learning metrics to test and evaluate *Hybroid*. To do so, we leverage a dataset that contains the original APK files and the mobile network traffic data generated by the applications. Next, we seek to identify the best detection classifier, and based on classifier performance, we try to use different parameter engineering. We compare our solution with other machine learning state-of-the-art related works, such as static and dynamic analyses based detection. The extracted results prove the advantages of our proposed solution, which combines the static and dynamic analysis of Android malware into a unified classification procedure.

3.1 Experimental Setup

We implemented the proposed methods using Python, Scikit-Learn, Tensorflow, and Keras. We set up our experiments on our Euklid server with 32 Core Processor, 128 GB RAM, and 16 GB GPU. Besides, we used 5-fold cross-validation. To obtain a reliable performance, we averaged the results of the cross-validation tests, executed each time with a new random dataset shuffle.

3.2 Evaluation Metrics

Due to the imbalanced nature of the dataset (see Section 3.3), accuracy may not be the only reliable indicator of classifier performance. Thus, the performance of detection and categorization will be evaluated with metrics such as precision, recall, and F-measure (F1-score). In general, the accuracy metric is used when true negatives and true positives are crucial; the F1-score is used when false positives and false negatives are more important. When the class distribution is nearly equal, accuracy can be used, whereas the F1-score is a better metric when we have imbalanced classes. However, in most real-life classification problems, the datasets are imbalanced, and therefore, the F1-score is a better metric to evaluate the model. However, since other related studies report accuracy as their primary evaluation metric, we also compare and consider accuracy as a comparison metric. Another metric to evaluate our work is to consider the receiver operating characteristic (ROC) curve, which presents the true positive rate (TPR) against the false positive rate (FPR).

Table 3. Dataset descriptions

Name	Number	Description	Distribution(%)
APK files	2,126	All program code files	100%
PCAP files	2,126	All the raw network traffic files	100%
Benign APKs	1,700	No. of benign APK	80%
Adware APKs	124	No. of Adware category APK	5.9%
Ransomware APKs	112	No. of Ransomware category APK	5.2%
Scareware APKs	109	No. of Scareware category APK	5.2%
SMSmalware APKs	101	No. of SMSmalware category APK	4.7%

3.3 Dataset

For the dataset, we use the public CICAndMal2017 [13]. The benign applications were collected from the Google play market published in 2015, 2016, and 2017. On the other hand, the malicious ones were collected from various sources such as VirusTotal[5] and Contagio security blog[6]. The dataset includes 426 malware and 1,700 benign samples with their corresponding network traffic raw data, which are delicately captured from physical smartphone devices while running the applications.

In the networking part, the phones' behavior was generated by scripts, which imitated normal phone usage like phone calls and utilized SMS along with GPS spoofing and web browsing. Every phone was also connected to a Gmail, Facebook, Skype, and WhatsApp account. The normal behavior of phones was captured in PCAP files that served as the entry point in our work. After infecting every phone with malware from the malware pool provided with the dataset in the form of APK files, the resulting network communication was collected. Table 3 provides a short description of the CICAndMal2017 dataset.

3.4 Power Law and Opcode Embedding

Before moving to our evaluation tasks, we use the distribution of our opcode to prove the reasonability of using natural processing language techniques in our works. In order to get the reasonable opcode2vec module, we pre-train the opcode2vec by the AndroZoo dataset. We extract all opcodes by the Androguard tool and obtain 18,240,542 opcodes in total. Then, we take those opcodes as our word corpus to train the opcode2vec model. Figure 5 presents the opcode distribution for the above datasets. More specifically, it shows Dalvik's opcode distribution, which has 216 opcodes, and the top-20 opcodes are presented. They all follow the power-law distribution, which makes borrowing word embedding techniques from natural language processing to do opcode embedding reasonable.

[5] `https://virustotal.com`
[6] `http://contagiominidump.blogspot.com`

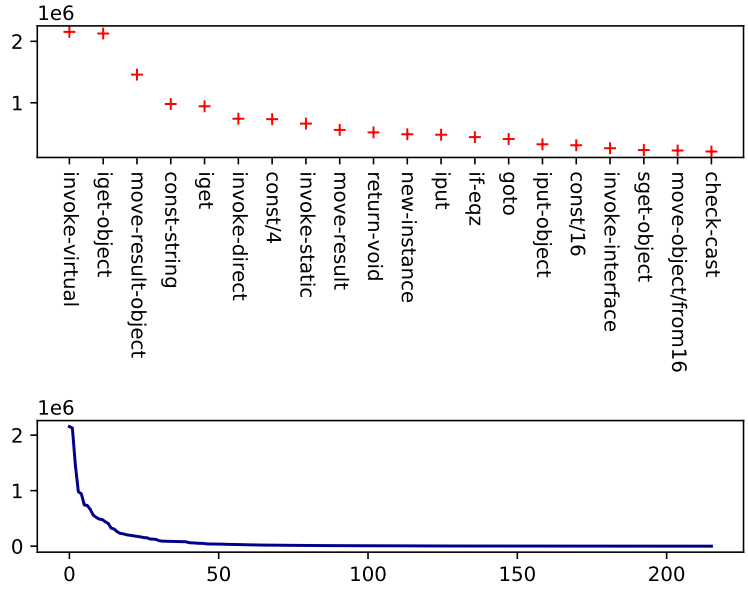

Fig. 5. Power-law distribution for Dalivk opcodes

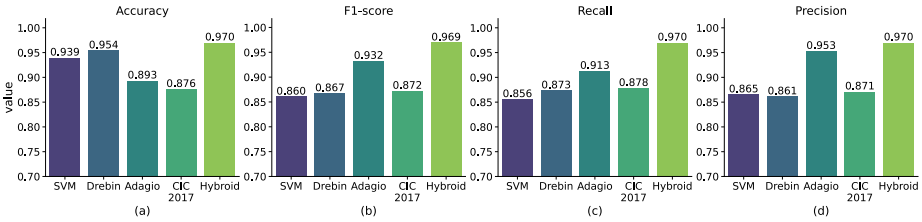

Fig. 6. Malware detection overall performance of different related works

3.5 Performance of Classifiers

In this section, the evaluation of the Android malware detection and categorization algorithms is presented in detail. For the malware detection experiments, we compare *Hybroid* with other related solutions. Figure 6 shows the difference between our solution and the other malware detection schemes. *Hybroid* demonstrates an accuracy of 97.0% (Figure 6-a), while CIC2017 [13], DREBIN [3], SVM [8], and Adagio [10] demonstrate accuracy of 87.6%, 95.4%, 93.9%, and 89.3%, respectively. Other metrics, such as F1-score, precision, and recall, are also presented in Figures 6-b, 6-c, and 6-d.

In addition, Figure 7 shows the ROC curve of our solution and the other compared algorithms, while TPR is plotted against the FPR for the various thresholds of the detection methods. As the ROC curve shows, *Hybroid* demonstrates the best performance (represented by the purple line), which means that

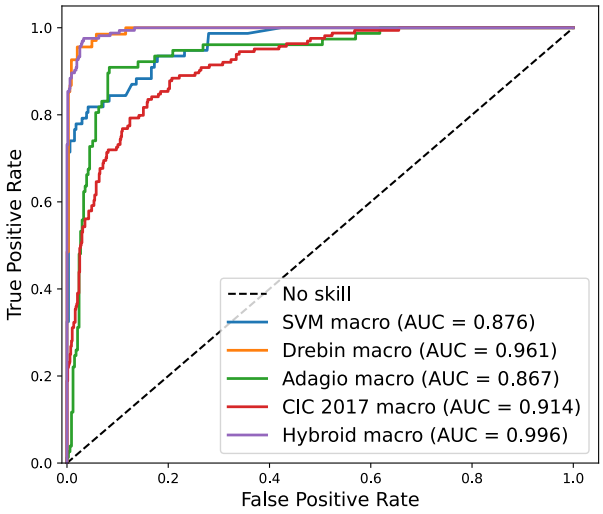

Fig. 7. Malware detection ROC curve of different related works

the combination of static and dynamic features boosts the classifier performance. Besides, we can observe, *Hybroid* presents the best area under the ROC curve (AUC), which is 99.6%, while the Adagio method shows the worst AUC of 86.7%. The Figure 7 also presents AUCs of the other compared solution classifiers.

To evaluate our work independently, we tested three various classifiers (decision tree, random forest, and gradient boosting) with static, dynamic, and combined features. We tested these three classifiers to identify differences in the performance of final classifiers. The obtained results confirmed that the combination of static and dynamic features yields the best performance. Moreover, we saw that the decision tree classifier demonstrates the lowest accuracy, precision, and recall compared to the other algorithms. Decision tree is also prone to overfitting. Random forest presented higher accuracy, precision, and recall as a more robust model than decision tree, limiting overfitting without substantially increasing error. On the other hand, compared with random forest, gradient boosting demonstrates the best metric results in our framework, implying that it is the most effective supervised learning algorithm for our experiment.

Gradient boosting is similar to random forest with a set of decision trees but with a main difference. It combines the results of week learners along the way, unlike random forest combines the results by majority rules or averaging at the end of the process. This accounts for the difference in the results.

Figure 8 shows the performance for the malware detection task. In the F1-score of various classifiers, we witness that combined features with gradient boosting achieve the best F1-score, which is 97%. Meanwhile, we only get 93% with static features from the program code and 95% with dynamic features from network flow. On the other hand, among the three classifiers with combined features, the gradient boosting classifier yields the best precision result, which

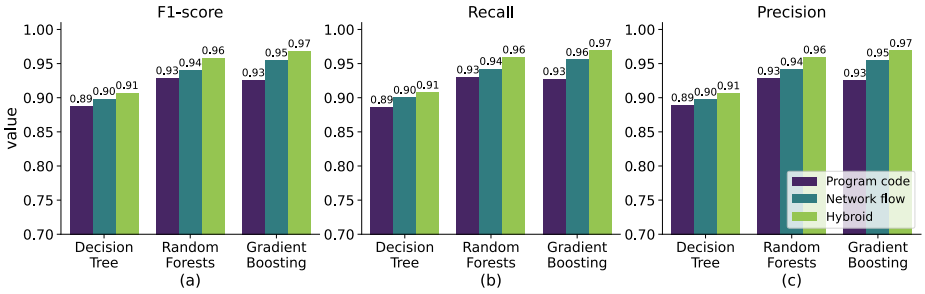

Fig. 8. Malware detection performance of the different classification algorithms

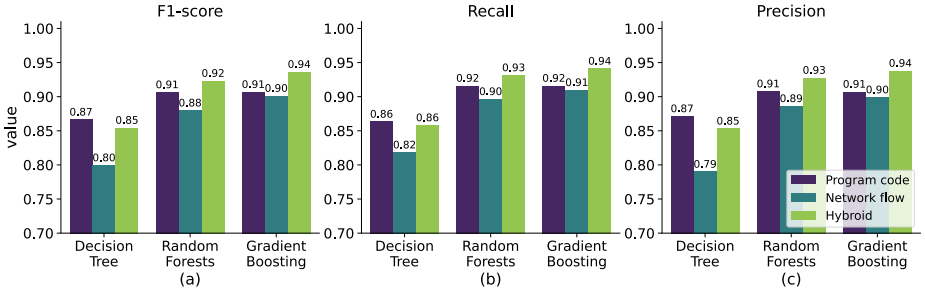

Fig. 9. Malware categorization performance of the different classification algorithms

is 97%. Meanwhile, random forest and decision tree demonstrate a precision of 96% and 91%, respectively.

Subsequent to the malware detection, we also evaluated *Hybroid* with the malware categorization, which is a multi-label classification task. We see from Figure 9 that the gradient boosting classifier receives the best results with combined features, namely 94% precision, 94% recall, and 94% F1-score. With the random forests classifier, our *Hybroid* also demonstrates significant results with a 92% F1-score. Figure 9 (a, b, c) depicts the evaluation results with F1-score, recall, and precision in detail.

Also, Figure 10 illustrates the ROC curves for the malware categorization task. Different curves show the different values of AUC. As it is shown, *Hybroid* demonstrates 97.6% macro accuracy on average for malware categorization, and categorization of the benign class receives the best performance AUC for 99.5%. For SMSware, we obtained the worst AUC, i.e., 94.6%. One potential reason for this issue could be the small number of SMSware in the dataset (only 4.7% samples are SMSware).

4 Limitation and Future Work

Although we combined static and dynamic analysis to improve the performance of *Hybroid*, some issues need to be addressed in the future. The biggest chal-

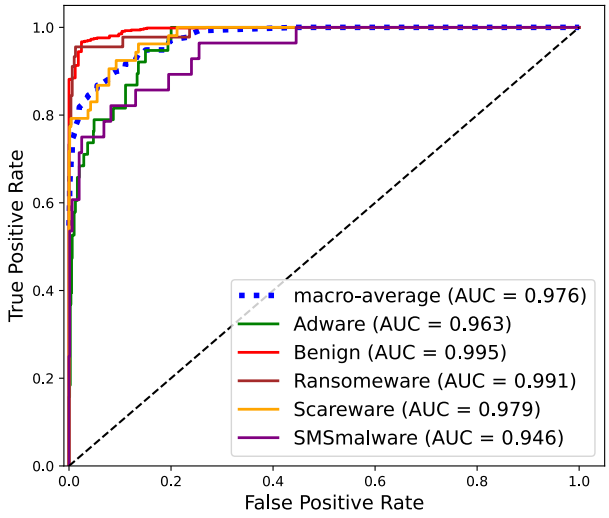

Fig. 10. Malware categorization ROC curve of gradient boosting

lenge is the lack of labeled data for CICAndMal2017 by Lashkari et al. [13]. They include only 426 malware and 1,700 benign APKs and their corresponding network traffic raw files. The main challenge is not having an alternative good-quality public dataset that covers the network traffic captured on real Android devices. For the malware detection, especially for the static feature-based work, the dataset with 2,126 samples is too small. However, for the networking dynamic feature-based work, 2,126 is a classic number. Actually, most networking dynamic analysis studies evaluate their frameworks with similar numbers, such as Jeon et al. [12] evaluate on 1,401 samples (1,000 malware and 401 benign) or Onwuzurike et al. [20] take 2,336 benign and 1,892 malware samples. Despite the static analysis work that needs more data samples, the number for networking dynamic analysis is normal. To address the lack of enough labeled data in the static analysis, we also separately trained and tested our static graph-based model with 45,592 malware and 90,313 benign samples following TESSERACT [23] policies (we split 80% of the whole dataset for training and the other 20% for testing). These data samples are captured from the AndroZoo[7], VirusShare[8], VirusTotal and we achieved the accuracy and F1-score of 95.0% and 96.0% respectively which shows that our methodology demonstrate competitive results on much larger dataset too.

Also, for the static analysis, *Hybroid* is affected by the obfuscated APKs, and we cannot successfully extract graph features from 47 obfuscated APKs. To further improve the robustness of *Hybroid*, we plan to extend CICAndMal2017

[7] `https://androzoo.uni.lu`
[8] `https://virusshare.com`

dataset in the near future to have more labeled network traffic data which are captured from Android real devices.

5 Related Work

Detecting Android malware and categorizing its families have attracted much attention from researchers as Android smartphones are gaining increasing popularity. Methods for Android malware detection are generally classified into traditional feature codes and machine learning.

For the traditional feature-based approaches, the detectors inspect the classical malicious behaviors. For example, program code-based malware detection methods extract features from the code itself. Technically, those features include permission [2,32], API call [1,3,22,32], N-gram [3], and CFG [10] based methods. Malware detection methods that use permissions and intents extract them from manifest files to detect Android malware [3]. In general, DREBIN performs a comprehensive static analysis, gathering as many application features as possible. These features are embedded in a joint vector space, such that typical malware patterns can be automatically identified and used to explain our method's decisions. In contrast with our work, DREBIN takes permissions and intents from manifest files, which cannot work for the obfuscated APKs. Meanwhile, *Hybroid* takes the graph structure from the program code, which obfuscation cannot affect. Additionally, we consider dynamic features from network flow, whereas DREBIN only considers the static features.

Graph-based malware detection systems use the graph structure for detection purposes, such as the Apk2vec [19] and the Adagio [10]. Adagio [10] shows a kernel-hashing-based malware detection system on the function call graph, which is based on the efficient embeddings of function call graphs with an explicit feature map inspired by a linear-time graph kernel. In an evaluation with real malware samples purely based on structural features, Adagio outperforms several related approaches and detects 89% of the malware with few false alarms, while it also allows for pinpointing malicious code structures within Android applications. Both the Adagio and our solution are based on the function call graph of Android applications. However, we design the graph embedding based on the function call graph, whereas Adagio uses the kernel-hashing method. In addition, we also take the network flow into our *Hybroid* to obtain the dynamic features.

Machine learning and deep learning techniques are also heavily introduced into the network traffic analysis. Researchers use manual indicated features to recognize a network traffic application pattern with traditional machine learning algorithms, such as traffic classification, network security, and anomaly detection [15,25,32]. Finally, for network traffic analysis, there are three different granularities: raw packet, flow, and session levels [13,16,24,25]. CICAndMal2017 [13] takes network traffic as the dynamic features to detect and categorize the Android malware. Compared to our work, it only considers the network flow rather than other static features, such as program code, permissions, and intents.

6 Conclusion

In this paper, we presented *Hybroid*, a layered Android malware classification framework, which utilizes network traffic as a dynamic and code graph structure as static behavioral features for malware detection. As a hybrid approach, it extracts not only 13 network flow features from the original dumped network dataset but also introduces NLP inspired technique based on function call graph embedding that converts the whole graph structure of an Android application into a vector. *Hybroid* utilizes the network flow features in combination with the graphs vectors to detect and categorize the malware. Overall, it demonstrates an average accuracy of 97.0% and 94.0% in detecting and categorizing the Android malware, respectively. The empirical results imply that our stated solution is effective in the detection of malware applications.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 830892 (SPARTA), No. 883275 (HEIR), and No. 833115 (PREVISION).

References

1. Aafer, Y., Du, W., Yin, H.: Droidapiminer: Mining Api-Level Features for Robust Malware Detection in Android. In: International conference on security and privacy in communication systems (2013)
2. Arora, A., Garg, S., Peddoju, S.K.: Malware Detection Using Network Traffic Analysis in Android Based Mobile Devices. In: International Conference on Next Generation Mobile Apps, Services and Technologies (2014)
3. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In: The Network and Distributed System Security Symposium (NDSS) (2014)
4. Bengio, Y., Courville, A., Vincent, P.: Representation Learning: A Review and New Perspectives. IEEE transactions on pattern analysis and machine intelligence **35**(8), 1798–1828 (2013)
5. Canfora, G., De Lorenzo, A., Medvet, E., Mercaldo, F., Visaggio, C.A.: Effectiveness of Opcode Ngrams for Detection of Multi Family Android Malware. In: International Conference on Availability, Reliability and Security (2015)
6. Chen, C., Liu, Y., Shen, B., Cheng, J.J.: Android Malware Detection Based on Static Behavior Feature Analysis. Journal of Computers **29**(6), 243–253 (2018)
7. Comparetti, P.M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., Zanero, S.: Identifying Dormant Functionality in Malware Programs. In: 2010 IEEE Symposium on Security and Privacy (2010)
8. Dai, G., Ge, J., Cai, M., Xu, D., Li, W.: Svm-Based Malware Detection for Android Applications. In: ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec) (2015)
9. Dai, H., Dai, B., Song, L.: Discriminative Embeddings of Latent Variable Models for Structured Data. In: International Conference on Machine Learning (ICML) (2016)

10. Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural Detection of Android Malware Using Embedded Call Graphs. In: ACM Workshop on Artificial Intelligence and Security (2013)
11. Haq, I.U., Caballero, J.: A Survey of Binary Code Similarity. arXiv preprint arXiv:1909.11424 (2019)
12. Jeon, J., Park, J.H., Jeong, Y.S.: Dynamic Analysis for IoT Malware Detection With Convolution Neural Network Model. IEEE Access **8**, 96899–96911 (2020)
13. Lashkari, A.H., Kadir, A.F.A., Taheri, L., Ghorbani, A.A.: Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification. In: International Carnahan Conference on Security Technology (ICCST) (2018)
14. Li, C., Mills, K., Niu, D., Zhu, R., Zhang, H., Kinawi, H.: Android Malware Detection Based on Factorization Machine. IEEE Access **7**, 184008–184019 (2019)
15. Malik, J., Kaushal, R.: CREDROID: Android Malware Detection by Network Traffic Analysis. In: ACM Workshop on Privacy-aware Mobile Computing (2016)
16. Marín, G., Caasas, P., Capdehourat, G.: DeepMAL – Deep Learning Models for Malware Traffic Detection and Classification. In: Data Science–Analytics and Applications (2021)
17. McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., et al.: Deep Android Malware Detection. In: ACM Conference on Data and Application Security and Privacy (CODASPY) (2017)
18. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed Representations of Words and Phrases and Their Compositionality. In: Advances in Neural Information Processing Systems (2013)
19. Narayanan, A., Soh, C., Chen, L., Liu, Y., Wang, L.: Apk2vec: Semi-Supervised Multi-View Representation Learning for Profiling Android Applications. In: 2018 IEEE International Conference on Data Mining (ICDM) (2018)
20. Onwuzurike, L., Almeida, M., Mariconti, E., Blackburn, J., Stringhini, G., De Cristofaro, E.: A Family of Droids-Android Malware Detection via Behavioral Modeling: Static vs Dynamic Analysis. In: Annual Conference on Privacy, Security and Trust (PST) (2018)
21. Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G.: Mamadroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). ACM Transactions on Privacy and Security (TOPS) **22**(2), 1–34 (2019)
22. Peiravian, N., Zhu, X.: Machine Learning for Android Malware Detection Using Permission and Api Calls. In: IEEE International Conference on Tools with Artificial Intelligence (2013)
23. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: TESSERACT: Eliminating Experimental Bias in Malware Classification Across Space and Time. In: USENIX Security Symposium (2019)
24. Taheri, L., Kadir, A.F.A., Lashkari, A.H.: Extensible Android Malware Detection and Family Classification Using Network-Flows and API-calls. In: International Carnahan Conference on Security Technology (ICCST) (2019)
25. Wang, W., Zhu, M., Zeng, X., Ye, X., Sheng, Y.: Malware Traffic Classification Using Convolutional Neural Network for Representation Learning. In: International Conference on Information Networking (ICOIN) (2017)
26. Xu, P., Eckert, C., Zarras, A.: Detecting and Categorizing Android Malware With Graph Neural Networks. In: ACM/SIGAPP Symposium on Applied Computing (SAC) (2021)

27. Xu, P., Eckert, C., Zarras, A.: Falcon: Malware Detection and Categorization With Network Traffic Images. In: International Conference on Artificial Neural Networks (ICANN) (2021)
28. Xu, P., Kolosnjaji, B., Eckert, C., Zarras, A.: Manis: Evading Malware Detection System on Graph Structure. In: ACM/SIGAPP Symposium on Applied Computing (SAC) (2020)
29. Xu, P., Zhang, Y., Eckert, C., Zarras, A.: HawkEye: Cross-Platform Malware Detection With Representation Learning on Graphs. In: International Conference on Artificial Neural Networks (ICANN) (2021)
30. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In: ACM SIGSAC Conference on Computer and Communications Security (CCS) (2017)
31. Yan, J., Yan, G., Jin, D.: Classifying Malware Represented as Control Flow Graphs Using Deep Graph Convolutional Neural Network. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2019)
32. Zulkifli, A., Hamid, I.R.A., Shah, W.M., Abdullah, Z.: Android Malware Detection Based on Network Traffic Using Decision Tree Algorithm. In: International Conference on Soft Computing and Data Mining (2018)