Why an Android App is Classified as Malware? Towards Malware Classification Interpretation

BOZHI WU, Nanyang Technological University, Singapore and Peking University, China
SEN CHEN*, College of Intelligence and Computing, Tianjin University, China and NTU, Singapore
CUIYUN GAO, Harbin Institute of Technology (Shenzhen), China
LINGLING FAN, College of Cyber Science, Nankai University, China and NTU, Singapore
YANG LIU, Nanyang Technological University, Singapore
WEIPING WEN, Peking University, China
MICHAEL R. LYU, Chinese University of Hong Kong, China

Machine learning (ML) based approach is considered as one of the most promising techniques for Android malware detection and has achieved high accuracy by leveraging commonly-used features. In practice, most of the ML classifications only provide a binary label to mobile users and app security analysts. However, stakeholders are more interested in the reason why apps are classified as malicious in both academia and industry. This belongs to the research area of interpretable ML but in a specific research domain (i.e., mobile malware detection). Although several interpretable ML methods have been exhibited to explain the final classification results in many cutting-edge Artificial Intelligent (AI) based research fields, till now, there is no study interpreting why an app is classified as malware or unveiling the domain-specific challenges.

In this paper, to fill this gap, we propose a novel and interpretable ML-based approach (named XMal) to classify malware with high accuracy and explain the classification result meanwhile. (1) The first classification phase of XMal hinges multi-layer perceptron (MLP) and attention mechanism, and also pinpoints the key features most related to the classification result. (2) The second interpreting phase aims at automatically producing neural language descriptions to interpret the core malicious behaviors within apps. We evaluate the behavior description results by leveraging a human study and an in-depth quantitative analysis. Moreover, we further compare XMal with the existing interpretable ML-based methods (i.e., Drebin and LIME) to demonstrate the effectiveness of XMal. We find that XMal is able to reveal the malicious behaviors more accurately. Additionally, our experiments show that XMal can also interpret the reason why some samples are misclassified by ML classifiers. Our study peeks into the interpretable ML through the research of Android malware detection and analysis.

CCS Concepts: • **Computing methodologies** → **Classification and regression trees**; • **Security and privacy** → *Software security engineering*.

Additional Key Words and Phrases: Android Malware, Interpretability, Machine Learning, Interpretable AI

*Sen Chen is the corresponding author. Email: chensen@ntu.edu.sg

Authors' addresses: Bozhi Wu, wubozhi@pku.edu.cn, Nanyang Technological University, Singapore and Peking University, China; Sen Chen, chensen@ntu.edu.sg, College of Intelligence and Computing, Tianjin University, China and NTU, Singapore; Cuiyun Gao, cygao@cse.cuhk.edu.hk, Harbin Institute of Technology (Shenzhen), Shenzhen, China; Lingling Fan, llfan@ntu.edu.sg, College of Cyber Science, Nankai University, China and NTU, Singapore; Yang Liu, yangliu@ntu.edu.sg, Nanyang Technological University, Singapore; Weiping Wen, weipingwen@pku.edu.cn, Peking University, China; Michael R. Lyu, lyu@cse.cuhk.edu.hk, Chinese University of Hong Kong, Hong Kong, China.

1049-331X/2020/1-ART1 $15.00
https://doi.org/10.1145/3423096

ACM Reference Format:
Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael R. Lyu. 2020. Why an Android App is Classified as Malware? Towards Malware Classification Interpretation. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2020), 29 pages. https://doi.org/10.1145/3423096

1 INTRODUCTION

Android malicious applications (malware) have become a serious security issue as the mobile platform has become increasingly popular [1]. For example, more and more app users store personal data such as banking transactions on their mobile devices [14, 15], consequently, hackers shift their attention on mobile devices and try to perform malicious behaviors through Android apps. It is not surprising that a number of approaches have been proposed for detecting Android malware. Specifically, traditional signature-based approaches [74, 76] require frequent updates of the signature database and fail to be effective in detecting emerging malware. Behavior-based approaches [34, 57, 62, 68] also rely on the predefined malicious behaviors, which is limited by the analysis of existing malicious samples. Data flow-based approaches [9, 24, 33, 42] are usually used to identify data leakage related malicious behaviors. Recently, researchers have proposed many effective Android malware detection methods by using a plethora of machine learning (ML) algorithms (e.g., KNN [6], SVM [7], Random Forest [53], and XGboost [29]) to classify and categorize malware. In these approaches, Android permissions and API calls are the commonly-used feature types [7, 16, 18, 63], and achieved a high detection accuracy (more than 90%). Meanwhile, researchers began to leverage deep neural networks like CNN and RNN (e.g., LSTM and GRU) to detect Android malware [26–28, 38, 72] and promising performance has been achieved.

However, these ML-based methods only provide a binary label to mobile users and app security analysts. In other words, these existing methods do not completely solve the problem of malware detection because they merely mean that the classified apps are most likely Android malware or benign apps. In practice, in many cases, only knowing the classification results is not enough. For example, (1) the app store needs to know exactly what malicious behaviors the apps employ, instead of classification results, in order to decide whether to remove them from markets. (2) For app security analysts, they need to identify various malware and then understand the malicious behaviors manually with substantial effort. It is a difficult and time-consuming task to analyze a large-scale dataset of Android malware in the wild. However, the truth is that millions of malware are classified and stored in the server. Therefore, interpreting and understanding what an ML model has learned and how the model makes prediction can be as important as the detection accuracy since it can guarantee the reliability of the classification model. Additionally, the robustness of ML models is facing the security threat of adversarial samples according to a large number of relevant research including Android malware [12, 16, 17, 19, 37, 40]. As ML-based methods are black-box and cannot explain how they make predictions, adversaries might fool these methods by constructing a little perturbation to misclassify malware as benign samples more smoothly.

In order to solve the problems mentioned above, we first investigated the approach of interpreting malicious behaviors in Drebin [7] and found that the approach localized malicious behavior from the trained model rather than the test sample itself. After that, we tried to explain the classification of the malware detection using an interpretable ML method called LIME [54], but the feature results are mismatched to the behavior because LIME did not consider the correlation between the input features. To take the correlation between different features into account, we find that attention mechanism has been applied in machine translation and computer vision (CV), and achieved great success of interpretability [8, 31, 66, 73]. Therefore, we follow this research line, and propose a novel and interpretable ML-based approach (named XMal) to detect Android malware and interpret how predictions are made. XMal leverages a customized attention mechanism with a multi-layer

perceptron (MLP) model, which pinpoints the key features most related to the prediction result, since the traditional attention mechanism cannot be used directly in Android malware detection scenario (§2.3). Apart from the binary result, it also automatically generates a descriptive explanation (i.e., a malicious behavior description) for the classification according to the key features. Additionally, it can help to explain why some benign apps are misclassified as malware and vice versa. We conduct comprehensive experiments to demonstrate its interpretability of Android malware detection, and the results show that XMal can detect Android malware effectively, with 98.35% accuracy, and can identify the malicious behaviors that are validated by a human study through an online survey. Our quantitative analysis can also be used to demonstrate the better performance on malware description generation of XMal. In addition, we compare the results with the state-of-the-art techniques in the interpretability of Android malware detection scenario. Finally, we present case studies and in-depth discussion about our approach.

In summary, we make the main contributions as follows.

- We are the first work focusing on the interpretability of Android malware detection and analysis. We concentrate on why an Android app is classified as malware rather than the detection accuracy only.
- We propose XMal to interpret the malicious behaviors of Android malware, by leveraging a customised attention mechanism with multi-layer perceptron (MLP).
- We conduct a human study by designing an online survey and a quantitative analysis to validate the capability of XMal regarding interpretability, and also provide an in-depth comparison study with the state-of-the-art techniques to demonstrate the effectiveness of XMal.
- We present several case studies and an in-depth discussion to highlight the lessons learned and the current status of interpretability of Android malware detection and analysis.

2 BACKGROUND

In this section, firstly we review several potential solutions for interpretability in Android malware detection and point out their weaknesses. Secondly, we introduce the attention mechanism as our work uses the concept of attention mechanism. Finally, we highlight the motivation of our work.

2.1 Potential Solutions for Interpretability in Android Malware Detection

ML technique is widely used to classify the samples into different categories, however without explaining the reason for the prediction results (i.e., not *interpretable*). *interpretable*, defined by Doshi-Velez et al. [23], is the ability to explain or present the results in understandable terms to human. In order to alleviate this problem, some general methods which are model-agnostic have been proposed, such as LIME [54] and LEMNA [36]. On the other head, researchers have also done some studies in areas of text categorization and image classification. For example, Arras et al. [8] tried to demonstrate that understanding text categorization can be achieved by tracing the classification decision back to individual words using layer-wise relevance propagation (LRP), a recently developed technique for explaining predictions of complex non-linear classifiers. Zhou et al. [73] proposed a new framework called Interpretable Basis Decomposition for providing visual explanations for image classification networks. By decomposing the input image into semantically interpretable components, the proposed framework can quantify the contribution of each component to the final prediction.

In Android malware detection and analysis, malware is identified by features (e.g., permissions, intents, and API calls) extracted from the APK file. Usually, app analysts first extract dangerous permissions and intents from AndroidManifest.xml. They utilize existing tools (e.g., dex2jar) to

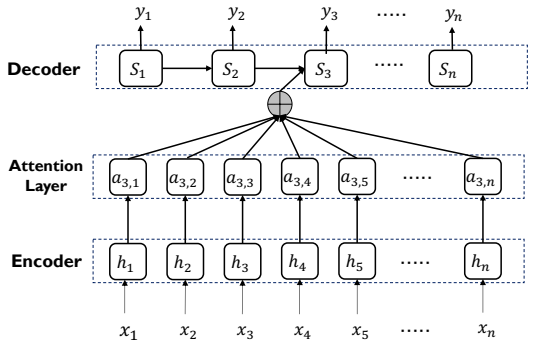


Fig. 1. Attention in machine translation

decompile Dalvik executable (dex) files in the Android application package (apk) file to get the source code and read the source code from the beginning to end to locate malicious code segments that lead to malicious behaviors. Finally, they can identify malware through malicious behaviors, which is very understandable to a human. In order to explain the predictions in ML, some key permissions, APIs, intents, or code segments should be used to match certain behaviors of Android apps, which help us understand what behaviors the Android app might perform, causing it to be classified as malware. Therefore, to explain why an app is classified as malware, we need to find out which features have a significant impact on the classification in ML, and whether they are indeed related to malicious behaviors of the malware. In order to do that, Drebin [7] utilized the simple detection function of linear SVM to determine the contribution of each individual feature to the classification result, which can be used to explain the classification of Android malware. However, since Drebin actually outputs the features with the highest weights in the ML classifier, rather than the test samples, the feature weights of different test samples are the same, which may be inaccurate. Melis et al. [48] proposed to leverage a gradient-based approach to identify the most influential local features. This method essentially obtains the gradient by approximating the original complex model, and there is inevitably a bias. In summary, there is no specific study on the interpretability of Android malware detection and analysis to interpret their corresponding malicious behaviors so far.

2.2 Attention Mechanism

Attention mechanism is a fairly popular concept and useful tool in the DL community in recent years [61]. In deep learning (DL), it refers to paying more attention to certain factors when processing data. It utilizes the attention vector to estimate how much an element is related to the target or other elements, and take the sum of their values weighted by the attention vector as the approximation of the target.

It was first proposed by Bahdanau et al. [10] to solve the problem of incapability of remembering long source sentences in neural machine translation (NMT). An attention layer is embedded between the encoder layer and the decoder layer, as shown in Fig. 1. The attention vector $c_i = \{a_{i,1}, a_{i,2}, a_{i,3}, a_{i,4}, .., a_{i,n}\}$ has access to the entire input sequence, which guarantees the ability of remembering long source sentences. More importantly, it also shows how significantly an input element is related to the output target, and which input element is more important or has a higher weight to generate the output.

Attention mechanism shows superiority in terms of classification and interpretability. It can help the model assign different weights to each part of the input, extract more critical and important

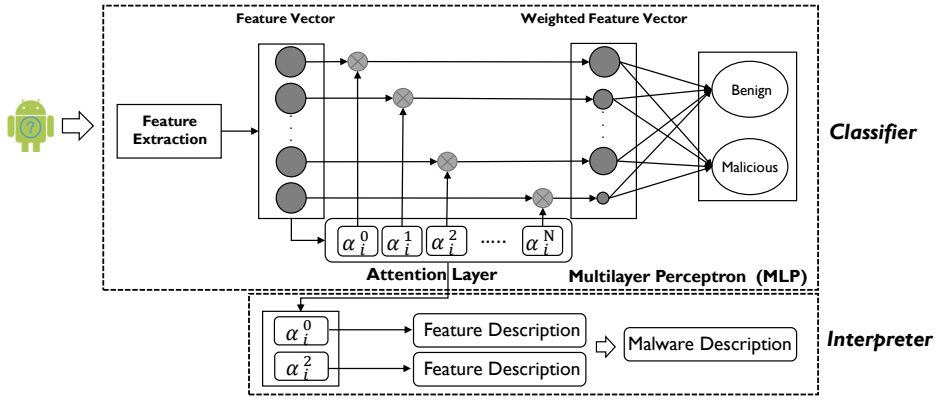


Fig. 2. Overview of our approach (XMal)

information, making the model's predictions more accurate, and make the prediction more understandable. For example, Xu et al. [65] proposed a method to explain why a certain word is output by visualizing the attention weights of the image region. This is why the attention mechanism is so popular. In this paper, we make the first attempt to use and customize attention mechanism in Android malware detection and analysis in order to interpret the prediction results.

2.3 Motivation of Our Work

In order to interpret the malware classification results, most existing interpretable ML-based methods utilize linear models or simple models (e.g., decision trees and linear regression) to approximate the original complex model [54], because these models can simply show the weight of each feature that contributes to the classification results. However, the usage of these models to approximate the original complex model inevitably introduces deviations. Additionally, most of these methods do not take into account the correlation between the input features. In fact, the features used by Android malware detection are usually highly correlated such as SmsManager.sendTextMessage and android.permission.SEND_SMS. This leads to the inability of these methods to give a correct explanation for Android malware detection. In order to address these problems and challenges, we propose a novel and effective method by using the attention mechanism with MLP for Android malware detection. The attention mechanism estimates how strongly a feature is correlated with other features and how important a feature is related to the prediction result. In Android malware detection scenario, we try to customize the attention mechanism through a fully connected network to learn the correlation between scalar-valued elements and assign corresponding weights to elements, since the traditional attention mechanism is performed on elements in the form of vectors and cannot be used directly in this case.

3 APPROACH

In this section, we first introduce the overview of our approach (named XMal), and then the details of each component.

3.1 Overview

As shown in Fig. 2, our approach (XMal) consists of two main components (i.e., a Classifier and an Interpreter). (1) The classifier component extracts API calls and used permissions from APK files as inputs, and aims at accurately predicting whether an app is malware. The classifier can also

pinpoint the key input features most related to the prediction result. (2) The interpreter component aims at automatically producing descriptions to interpret why an app is classified as malware. The behavior descriptions are generated through the rule-based method according to the documentation collected from Android Developers [32]. The details of each component are elaborated in Section 3.2 and Section 3.3, respectively.

3.2 Classifier Component

In this section, we introduce how to extract the key features that have more relevance to the classification results. The key feature extraction conducts two processes: feature extraction and model training. We detail the two processes as below.

3.2.1 Feature Extraction. Usually, if an Android app exhibits malicious behaviors, it will be granted with the necessary permissions and call the corresponding APIs. In fact, permission and API calls are the top two important and commonly-used feature types for Android malware detection and analysis [51]. A lot of studies used these two features as significant features for classifying Android malware, such as Drebin [7], DriodAPIMiner [6], DroidMat [63] and many other previous studies [16, 18, 29, 38, 47, 67, 70, 72]. Additionally, they contain semantics that can be used to help to understand the behaviors of the application. Therefore, in this paper, we follow the common practice and use API calls and permissions as the features to train a malware classifier. In Android system, there are hundreds of permissions, and the number of APIs exceeds 20,000. But not all of them are helpful in distinguishing malware. Li et al. [41] utilized three levels of pruning and found that only 22 permissions are significant for detecting malware. Therefore, we need to employ pruning to preserve those features that can be used to identify malware efficiently. Here we refer to the paper [16] and select 158 features (including 97 API calls and 61 permissions) for our study by using manual statistical pruning method in [16] from the original 2,114 features extracted from the training sample set. The selected features have a high degree of discrimination for malware classification, which is good for improving the accuracy and interpretability of the classification. Meanwhile, since API calls and permissions have more semantics that enable people to understand their role in the applications, using them as the features can help further interpret our model. Additionally, our approach is general that can be extended the new feature categories to capture more complex malicious behaviors according to new malware samples. In order to extract API calls and permissions, we utilize Androguard [22] to extract API calls and permissions from APK file, which are used to construct the feature vector. Here we denote a sample set by $\{\{(\boldsymbol{x_i}, y_i)\}|\boldsymbol{x_i} \in X, y_i \in Y, 1 <= i <= M\}$, where X is the set of $\boldsymbol{x_i}$ and Y is the set of y_i, $\boldsymbol{x_i} = (x_i^{(1)}, x_i^{(2)}, x_i^{(3)}, ..., x_i^{(N)})$ is the feature vector of the i-th sample, N is the total number of features, $y_i \in \{0, 1\}$ is the label of the i-th sample (i.e., 0 for benign, 1 for malicious), and M is the total number of samples. $x_i^{(j)}$ represents the j-th feature of the i-th sample. If the j-th feature exists in the i-th sample, then $x_i^{(j)} = 1$, otherwise, $x_i^{(j)} = 0$.

3.2.2 Customized Classification Model and Model Training. After extracting features and constructing a feature vector, we feed the feature vector to train the malware classifier. As shown in Fig. 2, the classifier consists of two layers: the *attention layer* and the *multi-layer perceptron (MLP)*. The attention layer is designed to learn weights of the features which can be regarded as relevancy scores between the features and classification results. Then the MLP maps the features weighted by the attention layer to the binary classification.

The traditional attention mechanism is to obtain the weight of the input feature by scoring how well the input feature and the output match, which can be formulated as follows:

$$\boldsymbol{e_{ij}} = score(\boldsymbol{s_{i-1}}, \boldsymbol{h_j}), \tag{1}$$

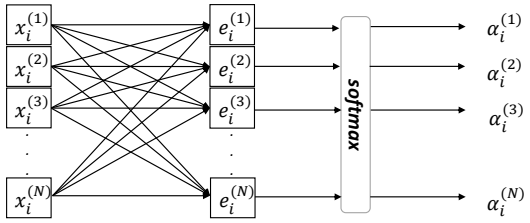


Fig. 3. Attention layer in XMal

where $\boldsymbol{s}_{i-1}$ is hidden state of output, and $\boldsymbol{h_j}$ is the j-th annotation of input. Then the feature weight can be computed by:

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{n} exp(e_{ik})}. \tag{2}$$

The score function will be different according to different scenarios. For instance, the score function in the paper by Luon et al. [46] is computed by:

$$score(\boldsymbol{s}_{i-1}, \boldsymbol{h_j}) = \boldsymbol{s}_{i-1}^{T} \boldsymbol{h_j}, \tag{3}$$

The input feature of traditional attention mechanism is generally expressed as a vector. But the features extracted from the samples are composed of scalar values. They can not be used to compute the score like Equation 3. Here we customize a fully connected network and a softmax function to implement the attention layer, as shown in Fig. 3. Because a fully connected network can capture the correlations between scalar-valued input features.

We compute how well all input features and the output at j-th position match by:

$$e_i^{(j)} = \sum_{k=1}^{N} x_i^{(k)} w_{kj}, \tag{4}$$

where w_{kj} is a learnable parameters of the fully connected network in attention layer. $e_i^{(j)}$ as the output at j-th position in the fully connected network, is a linear combination of all input features $x_i^{(k)}$. It can be regarded as the combination of a set of features that have different relevance to the input feature at j-th position. After the model training, the parameter w_{kj} will be assigned an appropriate value to show the correlation between the input feature at j-th position and other input features. Therefore, our customized attention layer has considered the correlation between the input features when computing the weight of input features.

Here we perform a softmax function on the output of the fully connected network to obtain the weights of input features at different positions. We denote attention vector by $\boldsymbol{\alpha_i}$, where $\boldsymbol{\alpha_i} = (\alpha_i^{(1)}, \alpha_i^{(2)}, \alpha_i^{(1)}, ..., \alpha_i^{(n)})$. $\alpha_i^{(j)}$ represents the weight of j-th feature in i-th sample, and is computed by:

$$\alpha_i^{(j)} = \frac{exp(e_i^{(j)})}{\sum_{k=1}^{N} exp(e_i^{(k)})}, \tag{5}$$

where $\alpha_i^{(j)}$ reflects the importance of the input feature at i-th position in deciding classification results.

After generating the attention vector through the attention layer, the MLP is used to map the features weighted by the attention vector to the binary classification. Here we denote the weighted

feature vector of the i-th sample by $\boldsymbol{c_i}$. It is obtained by weighting the input feature vector using the attention vector, and is computed by:

$$\boldsymbol{c_i} = \boldsymbol{\alpha_i} \boldsymbol{x_i}^T. \tag{6}$$

In the end, the classification result can be computed by:

$$\boldsymbol{y_i} = f(\boldsymbol{c_i}), \tag{7}$$

where $f(\cdot)$ represents the function of MLP that maps the input vector $\boldsymbol{c_i}$ into a binary prediction result.

When the training data are fed to train the classifier, the attention layer assigns different weights to the corresponding features based on their relevance to the classification result. Features that have more relevance to classification are assigned larger weights, while features with less impact are assigned smaller weights. Other interpretable ML methods aim to obtain the weight of the feature by approximating the original complex model. Unlike them, XMal directly obtains the weight of the feature by embedding the attention layer in the model, therefore there is no deviation. After feature extraction and model training, a malware classifier is generated. When a sample is input into the classifier, the classification result and a list of features with different weights are obtained. We remove those features that do not exist in the sample and sort the left features according to their weights. Then we select the top n features to generate the behavior description. Here, n is a hyperparameter. It is important to select a proper number for n. Although choosing more features as key features may help to identify more malicious behaviors, too many features will reduce the interpretability of classification [54]. The number for n is a heuristic value depending on concrete scenarios. According to the experiments, the default value is configured as 6.

Our customized model utilizes a fully connected network in the attention layer to capture the correlation between features, rather than a multi-layer fully connected network. A multi-layer fully connected network may capture much more complex relationships between features, but it is also difficult to understand and interpret since it involves too many mathematical operations, making it impossible for humans to follow the exact mapping from input feature to output. That is one reason why we do not use CNN or RNN models. In general, the deep learning models still cannot be interpreted accurately. How to interpret deep neural networks is an open challenge so far, which also belongs to our future work.

3.3 Malware Description Generation

In order to generate malicious behavior description for Android malware, we first match the malware key features to their corresponding semantics. We select a 158-dimensional feature vector as input to train the classifiers. We search Android developer documentation [32] for the semantics of each feature according to its name. The Android developer documentation has a detailed functional description for each API and permission. We download the detailed functional description of each feature. However, the functional descriptions include too many details and are difficult to understand comprehensively. We simplify and generalize them into simple semantics by intercepting and generalizing the key predicates, objects, and complements. For example, the functional description of READ_CONTACTS is "Allows an application to read the user's contacts data". We generalize it as "Collect contacts". Similarly, permission.READ_CONTACTS is generalized as "Collect contacts". After that, we use the feature and the corresponding semantic to build a semantic database (shown in Fig. 4). According to our observation, some features share the same semantics. For instance, URL.openconnection and URLConnection.connect share the same semantics of "Access the Internet". Besides, some features exhibit a similar functionality and can be combined

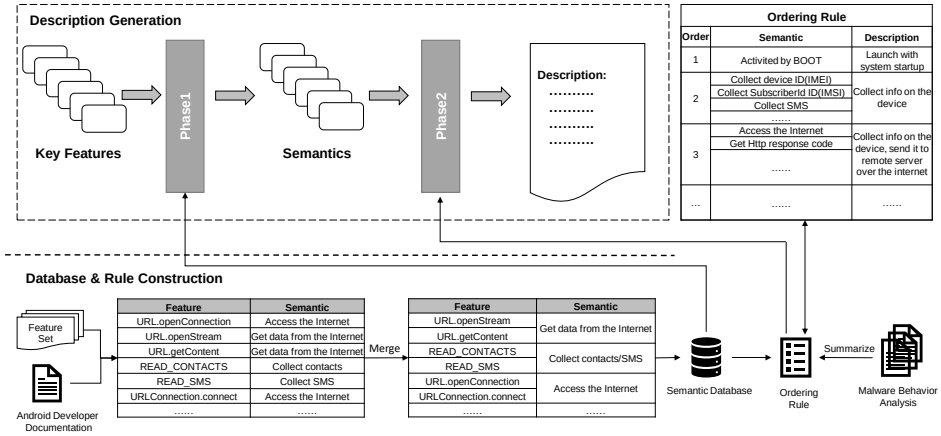


Fig. 4. Malware description generation

into one semantic feature. For example, permission.READ_CONTACTS and permission.READ_SMS are both about information collection and can be combined into "Collect contact/SMS". Therefore, we denote two rules as follows:

- ***Rule 1:*** *If features belong to a same functionality, they are assigned the same semantics.*
- ***Rule 2:*** *If features exhibit a similar functionality, they are assigned the similar semantics and the two similar semantics are combined into one.*

In this way, we match features with semantics based on their functional descriptions so as to obtain simple and useful semantics for features. After that, we convert the semantics into malware descriptions to make it easier for users to understand. In order to generate reasonable descriptions for the Android malware, we summarize ten basic malicious behaviors from a large number of malware, and establish the mapping relation between the malicious behaviors and their corresponding semantics. We also define some ordering rules to arrange the semantics (shown in Fig. 4, ordering rule) according to the malware behavior analysis by manual (shown in Fig. 4, malware behavior analysis). For example, if "activated by BOOT" exists, it should be ranked first; If "access the internet"and "collect IMEI" exist at the same time, "collect IMEI" should be in front of "Access the internet". Therefore, when "access to the Internet", "collect IMEI" and "Activate by BOOT" exist simultaneously, the order should be "activated by BOOT", "collect IMEI" and "access the Internet". Then they are converted into "Launch with system startup, collect info on the device, and send it to remote server over the Internet" through the mapping relation between semantics and malware behaviors.

Specifically, we first get a set of key features U, where $k_i \in U$ is the i-th key feature. Then we converted k_i into s_i one by one in the phase1 shown in Fig. 4, where s_i is the i-th semantics. According to Rule 1, if key features belong to a same functionality, they are assigned to the same semantics. Therefore, those semantics that exist in S are not added to S again. According to Rule 2, if the key features exhibit a similar functionality, their similar semantics are combined into one. Therefore, when the semantics s_i similar to semantic s in S appears, we combine it with s and then update s in S. After that, we convert the semantics into descriptions one by one in the phase2 shown in Fig. 4. Fig. 4 shows how the interpreter generates the malware description step by step and how the semantic database and ordering rule are established. The implementation details of the semantic database and ordering rules are provided on our website: https://sites.google.com/view/xmal/.

4 EVALUATION

In this paper, we aim to utilize the proposed method XMal to explain why an app is classified as malware. However, before interpreting the classification results, we should ensure that the detection accuracy is high enough since the malware detection accuracy is as important as the interpretability results, otherwise, the interpretation is meaningless. Therefore, in this section, we perform experiments to evaluate the malware detection accuracy and interpretability of the proposed method. Additionally, we also conduct an in-depth comparison study between XMal and the state-of-the-art techniques. We aim to answer the following research questions in our evaluation.

4.1 RQ1: What is the detection accuracy of XMal in the malware classification?

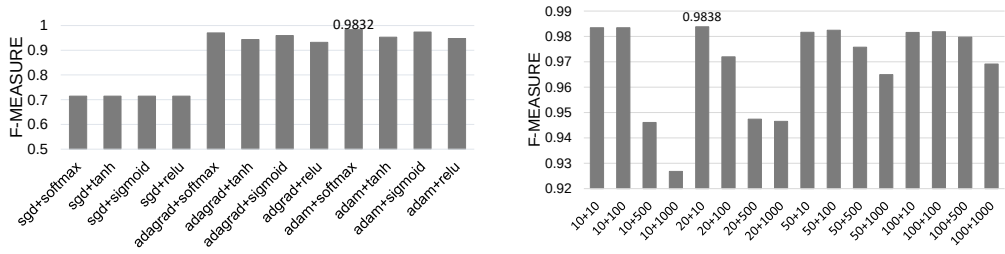
In this experiment, we first investigate the Android malware detection performance of XMal. We adapt the best hyperparameters of XMal for the best detection performance, and then conduct experiments to evaluate XMal and compare it with the state-of-the-art techniques. Finally, we investigate whether XMal can further be extended to the unsupervised Android applications in the wild.

4.1.1 Dataset. To conduct the experiment, we first collect a large amount of Android malware from two sources: 10,010 samples from the *National Internet Emergency Center* [2] and 5,560 samples from Drebin [7]. Most of the samples from the National Internet Emergency Center are the recent malicious samples rather than from old datasets such as Gemome [75] in 2011. These malware samples include a variety of threats for Android, such as data leakage, phishing, trojans, spyware, and root exploits. Apart from these malicious apps, we also fetched the top apps overall per category from Google Play Store and HUAWEI app store on July 2019 and collect 20,193 apps in total. We removed the ones that are classified as Android malware candidates by VirusTotal service [5]. Finally, we obtain 20,120 benign samples in total and 15,570 malicious samples, which are available on our website https://sites.google.com/view/xmal/.

4.1.2 Setup. To select the best hyperparameters for XMal, we first randomly split the 15,570 Android malware samples and 20,120 benign apps into a training set (70%, i.e., 24,983 samples in total) and a test set (30%, i.e., 10,707 samples in total). Note that these two sets have no overlap in our experiments. After that, we extract 158-dimensional feature vectors including 97 API calls and 61 permissions from the training set to train XMal, and utilize test set to evaluate the detection accuracy of XMal. Then we test different hyperparameters to XMal and finally determine the hyperparameters that can achieve the best detection performance.

In order to further demonstrate the superiority of XMal, we compare it with the state-of-the-art techniques and use *recall, precision, accuracy,* and *F-measure* to evaluate the detection performance. In this experiment, we compare XMal with Drebin [7] and LIME [54]. The reasons we select these two methods are as follows: 1) Drebin is a model-specific interpretable method like XMal, and achieve a high Android malware detection accuracy of 93.90%. 2) LIME proposes an effective model-agnostic method to interpret individual model prediction and obtain a high classification accuracy. It is one of the most valuable methods in model interpretability and has been cited and compared by many related research studies [36, 45, 49, 50, 55]. Comparing XMal with LIME is very significant for evaluating the interpretability and detection accuracy of XMal.

In conducting a comparison experiment, we first re-implement Drebin based on its published research paper. Since LIME is open-source, we are able to reuse it directly for our experiments. Specifically, for Drebin, we extract 422-dimensional features including API, permission, intent, activity, service, and hardware components from the dataset, and utilize them to train and test the model in Drebin. For LIME, since it is a model-agnostic method, we apply it to the MLP model. We



(a) Optimizer & Activation (b) Epochs & Batch_size

Fig. 5. Detection precision and recall under the different hyperparameters

Table 1. Detection results of three models

Models	Drebin	MLP in LIME	XMal
Recall	94.90%	97.13%	98.28%
Precision	95.94%	96.38%	98.48%
Accuracy	95.24%	96.50%	98.35%
F-measure	95.42%	96.75%	98.37%

extract the same features as XMal from the data set mentioned in the accuracy experiment to train and test the MLP model.

Moreover, to evaluate the detection accuracy of XMal in different malware families, we also select the top 16 malware families with the largest number of samples (cf. Table 2) according to the malware family tags provided by Drebin [7]. Since some malware families have too few samples to validate the interpretable results in the next experiments, we randomly select 10 samples for each malware family (i.e., 160 samples in total) from the test set for further investigation. In addition, in order to validate the detection accuracy of XMal in the samples from National Internet Emergency Center, we also randomly select 10 malicious samples from the test set. Finally, 170 malicious samples are selected to test and compare, we also randomly select 170 benign apps from the test set accordingly to validate XMal.

Parameter tuning for best classification performance. To achieve a better detection performance, we first search for the best hyperparameters (i.e., learning rate, optimizer, activation function, epochs, and batch_size) of XMal. Specifically, we set the learning rate to a set of values including 0.0001, 0.001, 0.01, and 0.1, which shows a little difference in detection performance. Therefore, we select 0.001 as the learning rate in our experiments. Fig. 5(a) demonstrates the detection results of applying different optimizer and activation function. As a result, "adam + softmax" achieves the best performance overall. We further investigate the impact of epochs and batch_size. As shown in Fig. 5(b), the configuration of 10 epochs and 20 batch_size achieves the best result.

4.1.3 Results. After determining the best hyperparameters of XMal, we use the test set (10,707 samples in total) to test XMal, Drebin, and LIME, and compare their detection performance. The experiment results are shown in Table 1. The result shows that the accuracy of the re-implemented model of Drebin is 95.24%, while the original accuracy in the paper is 93.90%, indicating the model we implemented is comparable to the original model. Note that, the accuracy of the re-implemented model is better than the original model because we perform the feature selection by

Table 2. Detection accuracy of the top 16 malware families

Families	TPR	Families	TPR	Families	TPR	Families	TPR
Adrd	100%	DroidKungFu	80%	BaseBridge	100%	Geinimi	100%
DroidDream	100%	GinMaster	100%	SendPay	100%	Iconosys	100%
FakeDoc	100%	Gappusin	100%	Plankton	100%	Kmin	100%
FakeInstaller	100%	MobileTx	100%	SMSreg	90%	Opfake	100%

using manual statistical pruning used in [16]. The accuracy of the three models are all above 95%, while XMal achieves 98.35% detection accuracy, outperforms the other two methods. Moreover, we evaluate XMal on the 170 benign test sample with a TNR (true negative rate) of 98.82%, which means that only 2 benign applications are misclassified as malware. We also test XMal on the 170 malware samples. The TPR (true positive rate) of the 10 malware samples from National Internet Emergency Center is 100%, and the TPR of each family is shown in Table 3. We can see that most malware families have a TPR of 100%, while DroidKungFu and SMSreg have TPRs of 80% and 90%, respectively, which means only two DroidKungFu malware and one SMSreg malware are misclassified as benign. In summary, XMal achieves high detection accuracy in malware detection. In order to validate XMal on unsupervised cases, we also randomly collect 1,000 Android apps from several Android application markets (e.g., Google Play Store, APKpure, coolapk, appchina, and mi.com) and employ XMal on these samples. We find that five of them are classified as malware. After manual analysis, we confirm that three apps[1] privately obtain the users' contacts and send them to a malicious remote server. Now they have been removed from the app stores. The other two apps[2] trick users into downloading apps and collect users' information. They still can be found on the website http://www.appchina.com/soft and we will report accordingly.

4.2 RQ2: How well does the malware description generated by XMal match the actual malicious behaviors of the malware?

We aim to interpret why an app is classified as malware in this section. To answer it, we conduct experiments to investigate whether the malware descriptions can match the actual malicious behavior of the malware.

4.2.1 Dataset. We perform interpretability experiments on all malicious samples (15,570 in total collected in Section 4.1.1) and generate the corresponding malware descriptions for each of them. In order to evaluate whether the malware descriptions generated by XMal match the actual malicious behaviors of the malware, we use the 170 malicious samples mentioned in Section 4.1.2 to establish the ground truth. Among them, since the 10 malware samples from National Internet Emergency Center have been analyzed before and the corresponding expert analysis has been validated by the expert team, we directly employ them as the ground truth of malware description. Note that, all members of the expert team are from National Internet Emergency Center and have engaged in malware analysis on the platform of Android and Windows for more than 3 years. They perform malware analysis and computer forensics on a daily basis, and are good at analyzing the malicious code/behaviors and identifying malware manually. For the other 160 samples from the top 16 malware families, we collect the corresponding expert analysis reports of each family from

[1]SHA1 values: 7BA69225D0B9B06DCADACA693DF58DE03228CDBE, AEDCB0B03C9193AC1F4B9CCFB31DDBA0FB7D9510, and AA0A1B157EA57E753C793F68141155A5A72F0620

[2]SHA1 values: 4246c467eb833805a0e7c09df0e8d72cf182bdfa and 8075a71fd8165fd1e33652fd7cd55f06b09a1697

Symantec [4] and Microsoft [3]. Meanwhile, we collaborate with the experienced expert team from National Internet Emergency Center, where they manually analyze these malware samples and provide the corresponding analysis reports. After that, we cross-validate the analysis reports from the two different resources and obtain the final ground truth of malware descriptions. Consequently, we can evaluate the interpretability results by comparing with the ground truth.

In addition, to further evaluate whether XMal can explain why the benign application is misclassified as malicious, we also select the 170 benign apps mentioned in Section 4.1.2 to conduct the experiments.

4.2.2 Setup. We first use XMal to generate the malicious behavior descriptions of the 15,570 malicious samples and evaluate if the generated descriptions (170 malware samples) match with the ground truth. In addition, to further evaluate whether XMal can explain why the benign application is misclassified as malicious, we employ XMal on the 170 benign samples to conduct more experiments. However, the evaluation may be biased by our subjective opinions. In order to mitigate this problem, we randomly select one sample from each malware family and conduct an online survey to investigate the quality of the malware description generated for these samples. Moreover, we also conduct a quantitative analysis to validate the effectiveness of XMal.

Evaluation Metrics. The ground truth and generated malware descriptions by XMal are unstructured text, which cannot be compared quantitatively. Inspired by Grounded Theory [21, 56], we extract "concepts" from the ground truth and generate malware descriptions, and compute how many "concepts" in the ground truth can be detected by XMal and how many "concepts" in the generated descriptions do not exist in the ground truth. Here, "concept" refers to a meta-behavior. For instance, "Activate when the mobile device is booted up" consists of two "concepts", "Activate" and "the mobile device is booted up". Here, we let *total_concepts* be the total number of "concepts" in the ground truth, *detect_concepts* be the number of "concepts" in the ground truth that are detected by XMal, and *surplus_concepts* be the number of "concepts" in the generated descriptions that do not exist in the ground truth. In order to quantitatively measure the interpretability results, we define the evaluation metric "interpretability result" (a.k.a. *ir*) as follows, and use *ir* to evaluate the generated descriptions of all malware by XMal.

$$precision = \frac{detect\_concepts}{detect\_concepts + surplus\_concept} \tag{8}$$

$$recall = \frac{detect\_concepts}{total\_concepts} \tag{9}$$

$$ir = \frac{2 \times precision \times recall}{precision + recall} \tag{10}$$

As the number of *detect_concepts* increases, *ir* becomes larger. When the number of *surplus_concepts* increases, *ir* becomes smaller. Therefore, the closer *ir* is to 1, the better the interpretability result. We take the *Adrd* in Table 5 as an example to illustrate the calculation process of *ir*. We extract concepts from the ground truth and the generated description and list them in Table 3. Specifically, "activate" vs. "launch" are the same concept. Similarly, "the mobile device is booted up" vs. "system startup", "access the Internet" vs. "over the Internet", "stead some info" vs. "collect info on the device", and "send to remote server" vs. "send it to remote server" are also the same concept. Consequently, we can know that *detect_concepts* is 5, *surplus_concepts* is 0, and *total_concepts* is 6. So *ir* is 0.91.

Parameter tuning for best interpretability. Before conducting the experiments, we perform hyperparameter tuning to select a proper value for *n*. We first obtain 50 malware samples and the corresponding expert analysis reports from National Internet Emergency Center. Then, we evaluate

Table 3. The "Concepts" of Adrd

	Concepts
Ground Truth	1. activate 2. the mobile device is booted up 3. access the Internet 4. download components 5. stead some info 6. send to remote server
Generated Description	1. launch 2. system startup 3. collect info on the device 4. send it to remote server 5. over the internet

Table 4. The *ir* computed by different values of *n*

n	1	2	3	4	5	6	7	8	9	10
ir	0.45	0.67	0.68	0.75	0.87	0.92	0.72	0.69	0.62	0.58

XMal and calculate *ir* of all samples under different values of *n* (i.e., ranging from 1 to 10). The results are shown in Table 4. We find that when *n* is set to 6, the average of *ir* for all samples is closest to 1, which is 0.92. Therefore, in the following experiments, *n* is set to 6. Note that 6 is not the best number of features in all scenarios. Hyperparameter tunning is necessary for different scenarios.

4.2.3 Results. We select one sample from each malware family and two samples (named "blackgame" and "xunbaikew1") from National Internet Emergency Center to demonstrate the interpretability of XMal. The interpretability results are as shown in Table 5. To illustrate how the experimental results explain why an app is classified as malware, we take Adrd and Opfake families as examples.

Android.Adrd is a Trojan horse in Adrd malware family that steals information from Android devices. As shown in Table 5, XMal outputs 6 key features (i.e., URL.openConnection, READ_PHONE_STATE, RECEIVE_BOOT_COMPLETED, requestLocationUpdates, getResponseCode, and getSubscriberId) for a sample of Adrd, and generates the corresponding semantics (i.e., "Access the Internet", "Collect IMEI/IMSI/location", and "Activate by BOOT") and malicious behavior description (i.e, *"Launch with system startup, collect info on the device, and send it to remote server over the internet"*). The malicious behavior description generated by XMal can clearly explain the reason why the sample of Adrd is classified as malware. In addition, the expert analysis of Adrd in Table 5 shows that it has the behavior of re-executing itself when the mobile device is booted up, stealing information and sending to a remote server. This is consistent with the malicious behavior description generated by XMal, which demonstrates the effectiveness of XMal. Additionally, we cross-validate through three co-authors to determine whether the semantics of the generated description by XMal is consistent with the ground truth (i.e., Expert Analysis). We accept the result only if all of us agree on it.

Opfake family sends SMS messages to premium-rate numbers on the Android platform. As can be seen from Table 5, XMal outputs four key feature (i.e., SEND_SMS, openConnection, READ_PHONE_STATE, and getNetworkOperator) for a sample of Opfake, and generates the corresponding semantics (i.e., "Send SMS messages", "Access the Internet", and "Collect IMEI") and malicious behavior description (i.e., "Send SMS to premium-rate numbers, collect info on the device, and send it to a remote server over the internet"). The malicious behavior description generated by XMal is also consistent with the expert analysis of Opfake shown in Table 5, which also accurately explains why the sample in Opfake is classified as malware.

In addition to the two examples above, the malicious behavior descriptions of the other samples also match the expert analysis as shown in Table 5. XMal provides a fairly reasonable explanation for the classification results. However, there are also some exceptions, such as a sample of FakeInstaller

Table 5. Part of the Interpretability Results of XMal. The full list can be found on our website https://sites.google.com/view/xmal/

	Key Features	Semantics Matching	Description Generated by XMal	Expert Analysis (Ground Truth)
Adrd	URL.openConnection READ_PHONE_STATE RECEIVE_BOOT_COMPLETED LocationManager.requestLocationUpdates HttpURLConnection.getResponseCode TelephonyManager.getSubscriberId	1. Access the Internet 2. Collect IMEI/IMSI/location 3. Activated by BOOT	Launch with system startup, collect info on the device, and send it to remote server over the internet	1. Activate when the mobile device is booted up. 2. Access the Internet and download components 3. Steal some info and send to remote server.
BaseBridge	SEND_SMS URL.openConnection READ_PHONE_STATE RECEIVE_SMS URLConnection.connect RECEIVE_BOOT_COMPLETED	1. Send SMS messages 2. Access the Internet 3. Collect IMEI/SMS 4. Activated by BOOT	Launch with system startup, send SMS to premium-rate numbers, collect info on the device, and send it to remote server over the internet	1. Send and receive SMS 2. Info is sent to remote server: a) Subscriber ID b) Device manufacturer/model c) Android OS version 3. Activate when the mobile starts
DroidKungFu	URL.openConnection READ_EXTERNAL_STORAGE READ_PHONE_STATE URLConnection.getURL URLConnection.connect RECEIVE_BOOT_COMPLETED	1. Access the Internet 2. Write to external storage 3. Collect IMEI 4. Activated by BOOT	Launch with system startup, download malware to SD card, collect info on the device, and send it to remote server over the internet	1. Steal sensitive info: IMEI number, device version, operating system version, etc. 2. Download files from remote computer or the internet.
FakeInstaller	SEND_SMS READ_PHONE_STATE RECEIVE_SMS READ_SMS TelephonyManager.getNetworkOperator WAKE_LOCK	1. Send SMS messages 2. Collect IMEI/SMS 3. Unlock phone	Send SMS to premium-rate numbers, collect info on the device, keep running in the background	1. Send the premium SMS 2. Receive commands from a remote server
Gappusin	URL.openConnection READ_PHONE_STATE RECEIVE_BOOT_COMPLETED NotificationManager.notify	1. Access the Internet 2. Collect IMEI 3. Activated by BOOT 4. Notify the info	Launch with system startup, collect info on the device, and send it to remote server over the Internet, send a notification as system	1. Post device info such as IMEI, IMSI, and OS version. 2. Download apps/disguises as system updates.
Opfake	SEND_SMS URL.openConnection READ_PHONE_STATE TelephonyManager.getNetworkOperator	1. Send SMS messages 2. Access the Internet 3. Collect IMEI	Send SMS to premium-rate numbers, collect info on the device, and send it to remote server over the Internet	1. Send SMS to premium-rate num. 2. Access info about network. 3. Check the phone's current state.
blackgame	URL.openConnection SEND_SMS RECEIVE_SMS WRITE_SMS TelephonyManager.getDeviceId TelephonyManager.getSubscriberId	1. Access the Internet 2. Send SMS messages 3. Collect SMS/IMEI/IMSI	Send SMS to premium-rate numbers, collect info on the device, and send it to remote server over the internet	1. Send SMS to premium-rate num. 2. obtain phone num and device info and upload it to the remote server.
xunbaikew1	SEND_SMS ContentResolver.query READ_CONTACTS	1. Send SMS messages 2. Collect contact info	Collect contact info on the device, and send SMS to premium-rate num	Collect contact info, and then send SMS with the app download link to all contacts.

Table 6. Two misclassified benign apps

Sample	Key Features	Semantics Matching
HiViewTunnel	permission.INTERNET WRITE_EXTERNAL_STORAGE URL.openConnection TelephonyManager.getDeviceId	1. Access the Internet 2. Write to external storage 3. Collect DeviceId
HwSpaceService	permission.INTERNET URL.openConnection WAKE_LOCK ContentResolver.query READ_PHONE_STATE	1. Access the Internet 2. Unlock phone 3. Collect SMS/IMEI

shown in Table 5. The malicious behavior description generated by XMal includes the behavior of sending SMS to premium-rate numbers, collecting information on the device, and keeping running in the background, however, the expert analysis only includes the behavior of sending the premium SMS. After manual analysis, we find that this sample indeed has the behavior of collecting information and keeping running in the background. Another sample is "xunbaikew1", which collects contact information and sends SMS message with the app download link to all contacts. XMal captures the malicious behavior of collecting contact information but misses the behavior of sending SMS message as sending SMS to a premium-rate number. Actually, some key features such as SEND_SMS can be mapped to different malicious behaviors in different scenarios like sending SMS with malicious download links. XMal may not be able to cover all the malicious behaviors only by mapping the key features. It can be improved by adding more other information from apps. Based on the expert analysis of the top 16 malware families, we find that 13 malware families (except FakeInstaller, FakeDoc, and SendPay) have the behavior of stealing information and sending it to a remote server over the internet, and 7 malware families have the behavior of sending SMS messages. Moreover, some of the information stolen by malware families is the same (e.g., IMEI, OS version, and device ID). We can conclude that the APIs and permissions used to perform malicious behaviors between different malware families are similar in Drebin dataset, which is consistent with our experimental results.

As aforementioned in Section 4.1.3, two benign apps are misclassified as malware and three malware samples are misclassified as benign. We attempt to analyze why they are misclassified according to the interpretable results of XMal. The two benign applications that are misclassified are HiViewTunnel and HwSpaceService, which are internal system applications for the HUAWEI phone. We can see in Table 6 that the two apps do use some suspicious permissions and APIs, causing them being classified as malware. In fact, they are just built-in system apps that use sensitive APIs and permissions. In this case, it is difficult for XMal to correctly distinguish malware, as the built-in system apps have the same features and behaviors as malware. For the three malware samples that are misclassified, XMal outputs no key features for all of them, which means that XMal does not identify any key features of these samples to classify them as malware, resulting in malicious samples being misclassified as benign. We further manually analyze these three samples and find that the malware APK file in SMSreg lacks the configuration file, AndroidManifest.xml, resulting in that the app has no permission to perform malicious behaviors so as to be identified as benign. The remaining two samples hide malicious behavior in the .so file and the asset folder, causing their malicious behaviors to be unrecognizable because XMal does not analyze the .so files and the files in the asset folder.

4.2.4 Online Survey. In order to alleviate the bias caused by subjective opinions, we conduct an online survey to investigate the respondents' evaluation of the interpretable results (i.e., malware description) generated by XMal.

Dataset. We randomly select one sample from each malware family and use the corresponding interpretability results of these malware to design the survey.

Participant Recruitment. We recruit 33 people from industrial companies and our universities to participate in the experiments via emails and word-of-mouth. Among the participants, 60.6% come from industry, and the rest come from academia. Note that, 6 security analysts from the National Internet Emergency Center also corroborate with us and help to accomplish this online survey. They come from different countries, such as USA, UK, Germany, China, Singapore, and Australia. They have a variety of occupations, ranging from PhD students, post-doctoral researchers, and professors. Fig. 6 shows the country and occupation distribution of participants. Their expertise includes app developers, computer security professionals, and machine learning researchers. Among them, 20

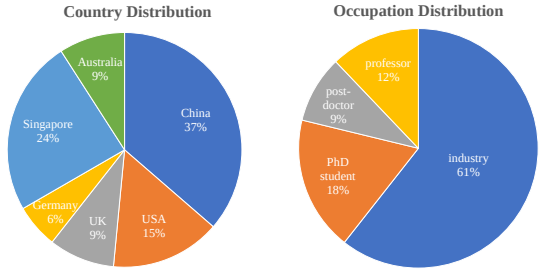


Fig. 6. Country and occupation distribution of participants

participants have experience in malware classification, while 17 respondents have more than 1 year of Android malware classification experience.

Experiment Procedures. We start the online survey with a brief introduction. We explain to the participants that our task is to evaluate how well the generated malware description results match the ground truth. Then the participants are required to provide their personal information relevant to the survey. In order to quantitatively measure the quality of the generated results, we define the rating scale as 1 to 5 where a higher score means that the generated results match the ground truth better. The participants are required to rate the quality of the generated results by comparing them with the ground truth of a total of 16 malware samples from 16 malware families.

There are two main tasks that participants are required to complete. Participants need to (1) fill in their personal information in the survey, such as name, country, academia or industry, field of work, the role at work, and their experience in Android malware classification, etc., and to (2) click the corresponding button in the survey to rate the generated malware descriptions of each sample by comparing them with the ground truth. The online survey contains 26 questions in total and takes about 20 minutes to complete. Table 7 demonstrates a part of the questions in the survey. The survey is available on https://forms.gle/RFUmPaSE9eKfG9zm8.

Survey Results. To ensure the quality of the survey result, we excluded those surveys that take less than 5 minutes to complete, and finally obtained 30 valid survey results. The average score of each sample is shown in Fig. 7. The average score of Kmin is 4, which means that the generated result of Kmin is good. Except for Gappusin, all other sampled scores are more than 3, which means that the generated description results are acceptable. All in all, the average score of all samples is 3.7. Therefore, we can conclude that the overall generated description result is better than Acceptable and close to Good.

Table 7. Part of the questions in the survey

	Questions	**Rating Scale (Score: 1-5)**
Part2	Q1: score the following generated description of "Adrd0". **Ground Truth:** 1. Activate when the mobile device is booted up. 2. Access the Internet and download componets 3. Steal the following info and send to a remote server. **Generated Result:** Launch with system startup, collect info on the device and send it to remote server over the internet. Q2: score the following generated description of "BaseBridge0"	1: Poor 2: Marginal 3: Acceptable 4: Good 5: Excellent

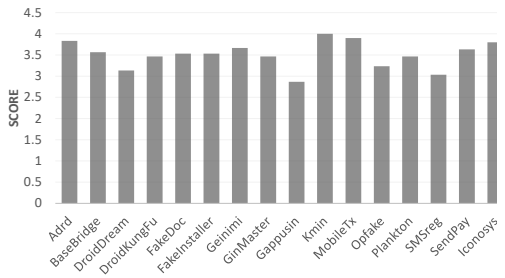


Fig. 7. The average score of all samples from survey

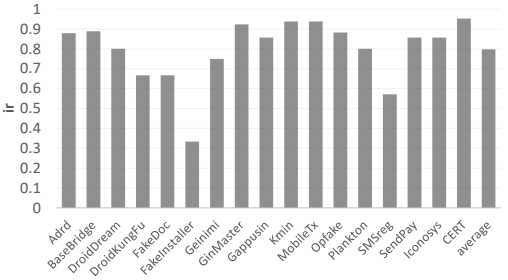


Fig. 8. The *ir* results of all family samples

4.2.5 Quantitative Analysis. In the end, besides the human evaluation, we further conduct a quantitative study to investigate how well the generated description matches the ground truth of the malware. Here, we define the 10 samples from the National Internet Emergency Center as the CERT family. The *ir* of interpretability results across all malware samples generated by XMal is shown in Fig. 8. We can see that the average *ir* of all malware samples is 0.80, while the *ir* of four families (i.e., GinMaster, Kmin, MobileTx, and CERT) is greater than 0.9. The *ir* of FakeInstaller and SMSreg families is lower than others, only 0.33 and 0.58 respectively. The reason is that most samples in FakeInstaller and SMSreg malware families have no more than two malicious behaviors, but XMal utilizes 6 key features to generate the malware description, resulting in a lot of *surplus_concepts* in these two malware families. To summarize, XMal achieves good interpretability results for all malware families except for FakeInstaller and SMSreg whose samples only have no more than 2 malicious behaviours.

4.3 RQ3: Does XMal achieve a better interpretation than the state-of-the-art techniques?

In this experiment, we aim to demonstrate that XMal can get a better interpretation than the state-of-the-art techniques. To achieve this goal, we conduct experiments on XMal, Drebin and LIME under the same data set, and compare their interpretation results through quantitative analysis and case studies.

4.3.1 Setup. In this experiment, to demonstrate the interpreting effectiveness of our XMal, we compare it with two state-of-the-art interpretable ML systems, Drebin [7] and LIME [54], on the 170 malicious samples selected and used in RQ2. The reasons we choose Drebin and LIME have been mentioned in Section 4.1.2. For Drebin, we obtain the weight of features by acquiring the coefficient of the liner SVM. The features with the highest weights are regarded as key features and used to interpret the classification result. Similarly, we perform LIME on the MLP model to obtain the feature weight of each test sample, and then select the feature with the highest weight as the key feature and use it to explain why an app is classified as malware or benign app.

In order to compare the three methods, we first calculate all key features for all samples and choose 6 key features in this comparison experiment. Note that the number of key features generated by LIME is 6 by default. For fairness, the number of key features generated by Drebin is also set to 6. We then obtain the corresponding semantics of the key features based on the semantic database, which allows us to understand the connection between the key features and the ground truth. Finally, we compare the interpretability of these three methods based on how well the key features and semantics match the expert analysis, and discuss the performance of the three methods based on the key features of the three malware families generated by them.

In addition, we compare the three methods from a quantitative perspective. We extract the "concepts" from the semantics of key features, and compute the *detect_concepts* and *surplus_concepts*. *total_concepts* can be extracted from the ground truth of malware samples. Then we respectively calculate the *ir* of the interpretation results generated by the three methods and make a comparison. Moreover, we compare the total number of key features across all these malware samples generated by the Drebin, LIME, and XMal.

4.3.2 Results. We compare the three methods on the 170 malware samples, and demonstrate the results of five representative malware (i.e., Adrd, GinMaster, MobileTx, blackgame, and xunbaikew1), as shown in Table 8. For the sample in Adrd, XMal outputs the key features, among which READ_PHONE_STATE, LocationManager.request and getSubscriberId match the behavior of collecting confidential information (e.g., location and IMEI/IMSI), and the features, openConnection and getResponseCode, match the behavior of sending information to remote location over the internet, the remaining feature, RECEIVE_BOOT_COMPLETED, matches the behavior of launching with system startup. By contrast, Drebin outputs the key features, among which getContent, getDeviceId, and openConnection match the behavior of stealing information and sending it to a remote location. However, the other features (i.e., Intent.action.MAIN, INSTALL_PACKAGES and NotificationManager.cancel) do not match the key malicious behaviors. The key features generated by LIME include openConnection, RECEIVE_BOOT_COMPLETED and getDeviceId, which can match the behavior of re-executing itself when the device is booted up and collecting information. But the remaining key features, like INSTALL_PACKAGES and ContentResolver.delete cannot match any behaviors of Adrd.

Similarly, for blackgame, xunbaikew1, and the samples in GinMaster and MobileTx, XMal outputs key features that match the malicious behavior of the corresponding sample. However, some of the key features generated by LIME and Drebin cannot match the behaviors of blackgame, xunbaikew1, GinMaster and MobileTx, as shown in the bold features in Table 8. For the sample in GinMaster, the key features generated by LIME can match most of the behavior of GinMaster, but the remaining features (i.e., NotificationManager.notify and NotificationManager.cancel) do not match any behaviors. For the sample in MobileTx, Drebin generates some key features that match the malicious behavior of stealing information and sending SMS messages to a premium-rate number, but the remaining features (e.g, Intent.action.MAIN and INSTALL_PACKAGES) do not match any malicious behavior of MobileTx. Blackgame and xunbaikew1 have similar phenomena with GinMaster and MobileTx.

For Drebin, the feature with maximum weight in Adrd, GinMaster, MobileTx, blackgame, and xunbaikew1 is always Intent.action.MAIN, and some key features can not reveal any malicious behaviors. The reasons are as follows. (1) Drebin utilizes the simple detection of linear SVM to determine the contribution of each individual feature to the classification and the feature weight of the model is only related to the model, but not to the test sample. If the features exist in test sample and the features have a large weight in the model, they will be selected as key features. Therefore, it makes sense that why Intent.action.MAIN is always the key feature and some key features generated by Drebin can not reveal the malicious behaviors. For LIME, as we can see, the key features generated by it do not match the behaviors of malware families very well. For example, LIME outputs the feature with maximum weight, i.e., NotificationManager.notify and NotificationManager.cancel, which do not match any malicious behaviors of GinMaster. The key features generated by LIME may not be accurate enough to give a reasonable explanation of the classification result in Android malware detection. (2) LIME generates a linear model to approximate the local part of the original complex model, which makes it difficult for LIME to accurately approximate the decision boundaries

Table 8. Comparison of three approaches (i.e., Drebin, LIME, and XMal). The bold texts refer to key features that cannot match the real malicious behavior. Ground Truth refers to the expert analysis corresponding to each sample.

	Drebin	LIME	XMal
Adrd	*Key Features:*	*Key Features:*	*Key Features:*
	Intent.action.MAIN	SEND_SMS	URL.openConnection
	INSTALL_PACKAGES	URL.openConnection	READ_PHONE_STATE
	URL.getContent	RECEIVE_BOOT_COMPLETED	RECEIVE_BOOT_COMPLETED
	TelephonyManager.getDeviceId	TelephonyManager.getDeviceId	LocationManager.request
	URL.openConnection	**INSTALL_PACKAGES**	HttpURLConnection.getResponseCode
	NotificationManager.cancel	**ContentResolver.delete**	TelephonyManager.getSubscriberId
	Corresponding Semantics:	*Corresponding Semantics:*	*Corresponding Semantics:*
	1.None 2.Install packege	1.Send SMS message 2.Access the Internet	1.Access the Internet 2.Collect phone status
	3.Get data from the Internet 4.Collect device ID	3.Activited by BOOT 4.Collect device ID(IMEI)	3.Activited by BOOT 4.Get updated location
	5.Access the Internet 6.Cancel notification	5.Install packege 6.Delete URI data	5.Get Http response code 6.Collect SubscriberId ID(IMSI)
	Ground Truth:1. Activate when the mobile device is booted up. 2. Access the Internet and download components. 3. Steal some info and send to remote server		
GinMaster	Drebin	LIME	XMal
	Key Features:	*Key Features:*	*Key Features:*
	Intent.action.MAIN	RECEIVE_BOOT_COMPLETED	URL.openConnection
	TelephonyManager.getDeviceId	TelephonyManager.getDeviceId	READ_PHONE_STATE
	TelephonyManager.getSimSerialNumber	**NotificationManager.notify**	RECEIVE_BOOT_COMPLETED
	URL.openConnection	URL.openConnection	HttpURLConnection.getResponseCode
	NotificationManager.cancel	TelephonyManager.getSimSerialNumber	TelephonyManager.getSubscriberId
	RECEIVE_BOOT_COMPLETED	**NotificationManager.cancel**	
	Corresponding Semantics:	*Corresponding Semantics:*	*Corresponding Semantics:*
	1.None 2.Collect device ID(IMEI)	1.Activited by BOOT 2.Collect device ID(IMEI)	1.Access the Internet 2.Collect phone status
	3.Collect ICCID 4.Access the Internet	3.Post notification 4.Access the Internet	3.Activited by BOOT 4.Get Http response code
	5.Cancel notification 6.Activited by BOOT	5.Collect ICCID 6.Cancel notification	5.Collect SubscriberId ID(IMSI)
	Ground Truth: 1. Steal info from the device. 2. Send info to remote server. 3. The malicious service is triggered when the device finishes a boot.		
MobileTx	Drebin	LIME	XMal
	Key Features:	*Key Features:*	*Key Features:*
	Intent.action.MAIN	SEND_SMS	SEND_SMS
	INSTALL_PACKAGES	TelephonyManager.getDeviceId	URL.openConnection
	URL.openConnection	RECEIVE_SMS	READ_PHONE_STATE
	RECEIVE_SMS	**INSTALL_PACKAGES**	TelephonyManager.getSubscriberId
	ActivityManager.restartPackage	READ_SMS	HttpURLConnection.getResponseCode
	SEND_SMS	**ActivityManager.restartPackage**	
	Corresponding Semantics:	*Corresponding Semantics:*	*Corresponding Semantics:*
	1.None 2.Install package	1.Send SMS message 2.Collect device ID(IMEI)	1.collect SMS 2.Access the Internet
	3.Get data from the Internet 4.collect SMS	3.Collect SMS 4.Install package	3.Collect phone status 4.Collect SubscriberId ID(IMSI)
	5.Break other applications 6.Collect SMS	5.Collect SMS 6.Break other applications	5.Get Http response code
	Ground Truth: 1. Steal info from the compromised device. 2. Send SMS messages to premium-rate number.		
blackgame	Drebin	LIME	XMal
	Key Features:	*Key Features:*	*Key Features:*
	Intent.action.MAIN	SmsManager.sendDataMessage	URL.openConnection
	WifiManager.setWifiEnabled	WifiManager.setWifiEnabled	SEND_SMS
	TelephonyManager.getDeviceId	**RECEIVE_MMS**	RECEIVE_SMS
	TelephonyManager.getSimSerialNumber	**ContentResolver.delete**	WRITE_SMS
	URL.openConnection	TelephonyManager.getNetworkOperatorName	TelephonyManager.getDeviceId
	NotificationManager.cancel	elephonyManager.getDeviceId	TelephonyManager.getSubscriberId
	Corresponding Semantics:	*Corresponding Semantics:*	*Corresponding Semantics:*
	1.None 2.Check whether wifi is enabled	1.Send SMS message 2.Check whether wifi enabled	1.Access the Internet 2.Send SMS message
	3.Collect device ID(IMEI) 4.Collect ICCID	3.Collect MMS 5.Collect network operator name	3.Collect SMS 4.Write SMS
	5.Access the Internet 6.Cancel notification	4.Delete URI data 6.Collect device ID(IMEI)	5.collect device ID(IMEI) 6.collect SubscriberId ID(IMSI)
	Ground Truth: 1. Send SMS message to premium-rate num. 2. Obtain phone num and device info and upload it to the remote server.		
xunbaikew1	Drebin	LIME	XMal
	Key Features:	*Key Features:*	*Key Features:*
	Intent.action.MAIN	SEND_SMS	SEND_SMS
	ContentResolver.query	ContentResolver.query	ContentResolver.query
	SEND_SMS	RECEIVE_SMS	READ_CONTACTS
	Runtime.exec	**INSTALL_PACKAGES**	
	READ_CONTACTS	READ_SMS	
	PowerManager.newWakeLock	**ActivityManager.restartPackage**	
	Corresponding Semantics:	*Corresponding Semantics:*	*Corresponding Semantics:*
	1.None 2.Query URL data 3.Send SMS message	1.Send SMS message 2.Query URL data	1.Send SMS message 2.Query URL data
	4.Execute command 5.Collect contacts	3.Collect SMS 4.Install package	3.Collect contacts
	6.Keep processor and screen awake	5.Collect SMS 6.Break other applications	
	Ground Truth: Collect contact info, and then send SMS with the app download link to all contacts.		

near an instance, especially in malware detection applications. For XMal, it generates key features that closely match the behaviors of the malware families.

We also conduct a quantitative analysis for the three methods. The *ir* of the interpretability results across all these malware samples generated by Drebin, LIME and XMal is shown in Fig. 9. XMal achieves the best interpretability results among the three methods across all these malware families. LIME is better than Drebin in most malware families except for *Gappusin* and *MobileTx*. XMal obtains the largest average value of *ir* across all malicious families. The total number of the

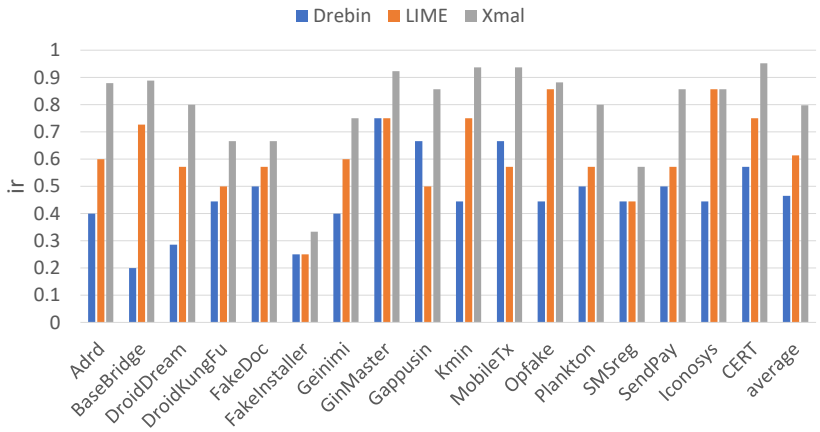


Fig. 9. The *ir* results across all malware samples generated by Drebin, LIME, and XMal

key features across all malware samples generated by the three method is shown in Fig. 10. Specifically, (1) openConnection is a common key feature for all families generated by the three methods, which indicates that most of malicious behaviors are based on Internet for these families. (2) Drebin and LIME output the same common feature TelephonyManager.getDeviceId for all families. This feature is used to get mobile information (e.g., IMEI). XMal outputs READ_PHONE_STATE with the similar function as getDeviceId. (3) Similarly, SEND_SMS is another common key feature for LIME and XMal, however, Drebin cannot identify SEDN_SMS for some malware families such as BaseBridge and Kmin. Both of them contain the behavior of sending SMS. The feature RECEIVE_BOOT_COMPLETED generated by LIME and XMal has the similar phenomenon with SEND_SMS. (4) Drebin generates two other common key features (i.e., Intent.action.MAIN and NotificationManager.cancel) for most families, but both of them cannot reveal malicious behaviors, as shown in Fig. 10 (green box). The similar phenomenon occurs on LIME for the feature NotificationManager.notify, as shown in Fig. 10 (red box).

In summary, Drebin generated some key features that cannot reveal malicious behaviors such as Intent.action.MAIN. LIME has a better performance in these families, but sometimes generates some key features that are meaningless to interpret the malicious behaviors in concrete cases (shown in Table 8). XMal generates key features for most malware families and is able to reveal the key malicious behaviors within apps. Therefore, XMal achieves a better performance on interpretability of Android malware detection.

5 DISCUSSION

In this section, we discuss the threats to validity, limitations of XMal and summarize open challenges in the interpretability of Android malware detection according to our study.

5.1 Threats to validity

Conclusion validity. Since we use the collected ground truth to validate the malware descriptions generated by our method, the results of XMal may differ if the ground truth is not accurate itself. To ensure the accuracy of the ground truth, we collaborate with an experienced expert team from National Internet Emergency Center and they manually analyze malware samples and provide the analysis reports. We also collect the corresponding expert analysis reports of each family from

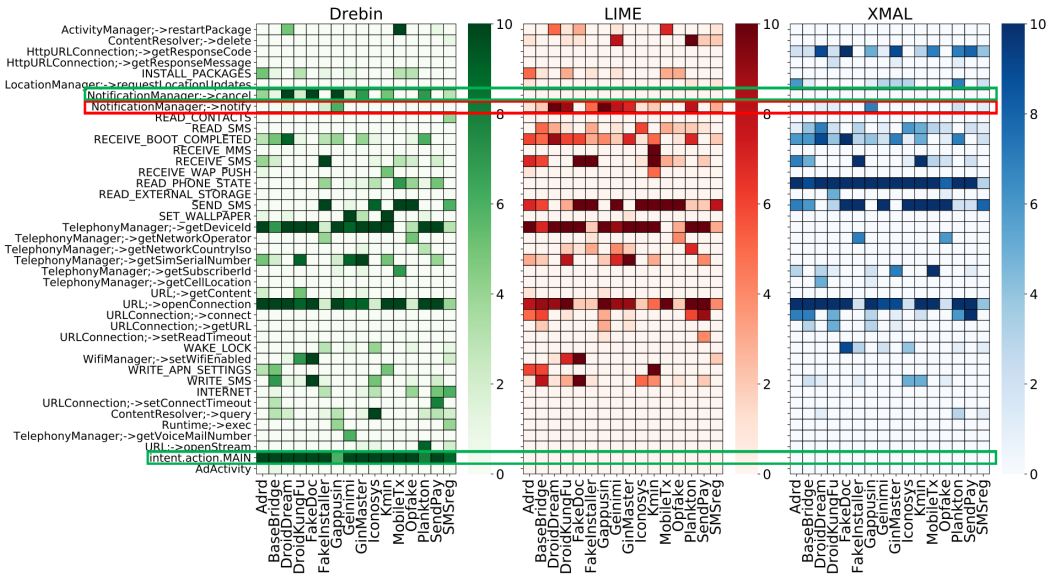


Fig. 10. Total number of key features across all malware samples generated by the Drebin, LIME, and XMal

Symantec and Microsoft, and cross-validate the analysis reports from the two different resources and obtain the final ground truth of malware descriptions.

Construct validity. We evaluate the generated results by manual comparison with the ground truth, the results may be biased by our subjective opinions. In order to mitigate this issue, we randomly select one sample from each malware family and conduct an online survey to investigate the quality of the generated malware description for these samples.

Internal validity. The performance of classification models depends on the training set. If the training set is small and not representative, the model cannot achieve a good generalization ability. We collect 15,570 malicious samples, including all varieties of threats for Android, such as data leakage, phishing, trojans, spyware, and root exploits. These malware samples are the recent malicious samples rather than the old dataset such as Genome [75] in 2011. Moreover, the performance of our method also depends on the hyperparameter configuration. It is important to select a proper value for n. We perform hyperparameter tuning and explain the procedures in Section 4.2.2.

External validity. Additionally, we simplify and generalize the functional descriptions into simple semantics by intercepting and generalizing the key predicates, objects, and complements. Therefore, semantics constitute is a threat to the external validity of the experiments. For reproducibility purpose, we release the functional descriptions and the constructed semantics on our website: https://sites.google.com/view/xmal/.

5.2 Limitations of XMal

We have proven the good performance of XMal in classification accuracy and interpretability, but it still has some limitations as follows. A malware sample can contain several different malicious behaviors. For example, a sample in the family, Geinimi, may collect information (e.g., IMEI, location, SMS messages, and contact) and upload them to a remote server, send SMS messages to a premium-rate number, install or uninstall software and create a shortcut. XMal cannot output features

that match all the malicious behaviors in the sample, because it makes predictions by focusing on the features with the highest weights (e.g., sending SMS messages to premium-rate number), causing it to only notice the malicious behavior of a certain part. It may be possible to improve it by using a multi-attention mechanism, which is our future work. To mitigate this issue, it may be helpful to use multi-attention to focus on different parts of features or different types of features. Multi-attention [38] utilizes a multi-modal deep learning method to learn various kinds of features, and uses multiple attentions to focus on more features and behaviors in order to identify more malicious behaviors.

The work in this paper is to explain why an app is classified as malware based on APIs and permissions. Although these two features can effectively target malicious behaviors and explain the classification result, they are not enough to explain how the entire malicious behaviors are implemented. More features should be taken into consideration. For example, If malware attempts to activate itself when the mobile device is booted up, it first has to register an intent.action.BOOT_COMPLETED intent-filter and apply for the RECEIVE_BOOT_COMPLETED permission in AndroidManifest.xml file, and then wait to receive a RECEIVE_BOOT_COMPLETED intent sent by the Android system in order to launch with system startup. In this case, Intent is a key feature to explain how malicious behaviors are implemented. Therefore, Intent should be taken into consideration in the analysis. However, considering more features might also result in a decrease in interpretability. When we obtain more key features, it might be more difficult to interpret the classification results. Therefore, it is also an important task to select reasonable features and make a trade-off between the number of features and interpretability performance.

Moreover, a small number of advanced malware try to hide their malicious behaviors by using native and reflective calls [30]. Our method can only analyze the APIs that access native and reflective calls to determine whether they are malicious. It is difficult to detect their malicious behaviors when the malicious payloads are only introduced by native code. Although our method did not use the native and reflective calls as features, nevertheless, it can still achieve a high detection accuracy of 98.37%. Besides, native and reflective calls do not have the developer documentations and a detailed functional description like permissions and APIs. As a result, we cannot construct their semantics for interpretability purpose. We therefore do not take these features into account in the feature set.

5.3 Open Challenges in Interpretability of Malware Detection

Many open challenges exist in explaining why an app is classified as malware. (1) One of them is a complex scenario. Some dangerous APIs and permissions may be used in benign apps for good purposes, such as the internal system apps. It is a great challenge for approaches that are based on features to predict and interpret. Features that are used in different scenarios may have different purposes. For example, in Section 4.1.3, two benign apps are misclassified as malware because they have dangerous API calls and permissions and are considered to perform malicious behaviors. But in fact, they are internal system apps, which own similar features and perform similar behaviors, such as monitoring the phone status. (2) Another challenge is the malicious behaviors of current malware become more complex. Malware may hide their behaviors through code obfuscation [43] and evade malware detection by downloading the payload after installation. For example, samples in fakeInstaller try to avoid analysis through code obfuscation and recompilation. The malware author modifies its DEX file with an obfuscated version of the recompiled code and uses anti-reverse techniques to avoid dynamic analysis and prevent malware from running in the emulator. Even with manual analysis, it is difficult to fully understand all the malicious behaviors of some complex malware as we need to analyze more code, API calls, permissions, or other features to locate and explain malicious behavior. However, it seems that the current interpretable machine learning

methods only use a small portion of features to explain the malicious behaviors. There is still a long way to go to explain why an app is classified as malware for all malicious samples.

6 RELATED WORK

6.1 Machine Learning-based Android Malware Detection

Since the traditional malware detection methods cannot handle an increasing number of malicious apps [13, 58], machine learning methods have become very popular and have achieved great success in Android malware [6, 7, 16, 18, 19, 25, 29, 41, 53, 64, 70]. For example, Aafer et al. [6] proposed to train a KNN classifier by learning relevant features extracted at API level and achieved accuracy as high as 99% with a false positive rate as low as 2.2%. Yerima et al. [70] presented a method to detect Android malware based on Bayesian Classification models obtained from API calls, system commands and permissions. Wu et al. [64] adopted the k-nearest neighbour classification model that leveraged the use of data-flow APIs as classification features to detect Android malware. Li et al. [41] utilized three levels of pruning by mining the permission data to identify the most significant permissions and trained an SVM classifier with 22 significant permissions. Other machine learning algorithms such as SVM [7], Random forest [53], and XGboost [29] were also used to detect malware and have proven to be effective.

With the popularity of deep neural networks, people began to utilize the deep neural network models for malware detection [27, 38, 47, 67, 71]. Yu et al. [71] proposed to train a malware detection model by using a representative machine learning technique, called ANN. McLaughlin et al. [47] proposed a malware detection system that used a deep convolutional neural network to learn the raw opcode sequence from a disassembled program. Kim et al. [38] utilized a multi-modal deep learning method to learn various kinds of features in order to maximize the benefits of encompassing multiple feature types. Xu et al. [67] used a Long Short Term Memory on the semantic structure of Android bytecode and applied Multi-layer Perceptron on the XML files in order to identify malware efficiently and effectively. All these method focused the malware detection accuracy rather than the malware interpretability.

6.2 Machine Learning Interpretability

People would like to interpret the machine learning models through visualization and behavior interpreting, which is what we are going to introduce.

6.2.1 Visualization. Visualization plays an important role in interpreting the machine learning algorithm, especially dimensionality reduction, clustering, classification and regression analysis. Elzen et al. [59] proposed a system that provided an intuitive visual representation of attribute importance within different levels of the decision tree, helping users to gain a deeper understanding of the decision tree result. Park et al. [52] utilized a simple graphical explanation to interpret the naive Bayesian, linear support vector machine and logistic regression classification process, and provided visualization of the classifier decisions and visualization of the evidence for these decisions. Krause et al. [39] proposed to visualize the ranking information of predictive features to help analysts understand how predictive features are being ranked across feature selection algorithms, cross-validation folds, and classifiers. Visualization can be used to provide an intuitive visual way to understand machine learning algorithms, but it is a better way to understand malware through malicious behaviors. Therefore, in this paper, we try to interpret machine learning algorithms through another way, behavior interpreting.

6.2.2 Behavior Interpreting. In order to interpret machine learning models itself, it is crucial to understand how they make predictions, which we define as behavior interpreting here. Through behavior interpreting, we can understand the relation between the input elements and models'

output. To achieve this goal, many researchers have tried to combine the elements that have the greatest impact on predictions to explain behaviors. In 2016, Ribeiro et al. [54] proposed a model-agnostic method called LIME. It treated the model as a black-box and then generated a linear model to approximate the local part of the model. The authors achieved this purpose by minimizing the expected locally-aware loss. After that, the authors tried to interpret the machine learning result through several features with the most weight. However, because LIME assumes that features are independent, although LIME is designed for explaining the predictions of any classifier, it actually supports CNN to work with image classifiers, but does not well support RNN and MLP. For malware detection, features are interrelated, which makes it difficult for LIME to accurately approximate the decision boundary near an instance. In 2018, Guo W et al. [36] proposed LEMNA, a high-fidelity explanation method that solves the problem in LIME. LEMNA utilized fused lasso, which acts as a penalty term that manifests as a constraint imposed upon coefficients in loss functions, to handle the feature dependency problems. Then, it integrated fused lasso into a mixture regression model to more accurately approximate locally nonlinear decision boundaries to support complex deep learning decision. The mixture regression model is a combination of multiple linear regression models. This method also interpreted the model through features with the most weight and is more fidelity than other existing methods. However, there are inevitably deviations due to the use of linear or simple models to approximate the original complex model. Apart from the above work, some survey papers [35, 44] also conducted studies on interpretability. All in all, they cannot interpret models' output accurately in Android malware detection. To solve this problem, we propose an interpretable machine learning model with a customized attention mechanism.

6.3 Applications of Attention Mechanism

The attention mechanism is mainly applied to machine translation and computer vision. Bahdanau et al. [10] first proposed to solve the problem of incapability of remembering long source sentences in neural machine translation (NMT). Xu et al. [65] inspired by the attention mechanism in machine translation, proposed an attention-based model that applied the attention mechanism to images to automatically describe the content of images. They first use a convolutional neural network to extract L feature vectors from the image, each of which is a D-dimensional representation corresponding to a part of the image. Then they use an LSTM decoder to consume the convolution features in order to produce descriptive words one by one, where the weights are learned through attention. The decoder selectively focuses on certain parts of an image by weighting a subset of all the feature vectors. The visualization of the attention weight can indicate the regions of the image that the model pays attention to in order to output a certain word. In addition, it also allows us to understand why some mistakes were made by the model. Vaswani et al. [60] proposed a new simple network architecture, the Transformer, based solely on the attention mechanism to perform machine translation tasks, and achieved good performance. There are many other applications for attention mechanism, such as machine reading [20], video summarization [11] and document classification [69]. Attention mechanism has been used to accomplish many machine learning tasks and achieved great success. Therefore, we make the first attempt to apply it in malware detection and interpret the classification results, but the traditional attention mechanism cannot be used directly since its elements and targets are expressed in vector form. We customize the attention mechanism through a fully connected network to learn the correlation between scalar-valued feature elements and assign corresponding weights to the elements.

7 CONCLUSION

In this paper, we proposed a novel approach called XMal to interpret the malicious behaviors of Android apps by leveraging a customized attention mechanism with the MLP model. XMal achieved a high accuracy in Android malware detection, and output a reasonable natural language description to interpret the malicious behaviors by leveraging the key features pinpointed by the classification phase. Additionally, we compared XMal with LIME and Drebin, and demonstrated that XMal obtained better performance in interpretability than the other two methods. Finally, we presented an in-depth discussion to highlight the lessons learned and open-challenges in this research field. The source code is released on the website https://github.com/wubozhi/Xmal.

ACKNOWLEDGMENTS

This work was supported by Singapore Ministry of Education Academic Research Fund Tier 1 (Award No. 2018-T1-002-069), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018 NCR-NCR005-0001), the Singapore National Research Foundation under NCR Award Number NSOE003-0001, NRF Investigatorship NRFI06-2020-0022, the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE004-0001, the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210717 of the General Research Fund), and the National Natural Science Foundation of China under project No. 62002084.

REFERENCES

[1] 2016. Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2016. http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/
[2] 2020. CERT. https://share.anva.org.cn/web/publicity/listMalware
[3] 2020. Microsoft. https://www.microsoft.com/en-us/wdsi/threats/
[4] 2020. Symantec. https://www.symantec.com/
[5] 2020. Virustotal. https://www.virustotal.com/gui/home/upload
[6] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in Android. In *International conference on security and privacy in communication systems*. Springer, 86–103.
[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of Android malware in your pocket.. In *Ndss*, Vol. 14. 23–26.
[8] Leila Arras, Franziska Horn, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. 2017. " What is relevant in a text document?": An interpretable machine learning approach. *PloS one* 12, 8 (2017), e0181142.
[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 259–269.
[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[11] Manjot Bilkhu, Siyang Wang, and Tushar Dobhal. 2019. Attention is all you need for Videos: Self-attention based Video Summarization using Universal Transformers. *arXiv preprint arXiv:1906.02792* (2019).
[12] Guangke Chen, Sen Chen, Lingling Fan, Xiaoning Du, Zhe Zhao, Fu Song, and Yang Liu. 2021. Who is Real Bob? Adversarial Attacks on Speaker Recognition Systems. *IEEE Symposium on Security and Privacy* (2021).
[13] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. 2019. GUI-Squatting Attack: Automated Generation of Android Phishing Apps. *IEEE Transactions on Dependable and Secure Computing* (2019).
[14] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An Empirical Assessment of Security Risks of Global Android Banking Apps. In *Proceedings of the 42st International Conference on Software Engineering*. IEEE Press, 596–607.
[15] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 2018. Are mobile banking apps secure? what can be improved?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 797–802.
[16] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security* 73 (2018), 326–344.

[17] Sen Chen, Minhui Xue, Lingling Fan, Lei Ma, Yang Liu, and Lihua Xu. 2019. How can we craft large-scale Android malware? An automated poisoning attack. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. IEEE, 21–24.

[18] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. 2016. Stormdroid: A streaminglized machine learning-based system for detecting Android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 377–388.

[19] Sen Chen, Minhui Xue, and Lihua Xu. 2016. Towards adversarial detection of mobile malware. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*. ACM, 415–416.

[20] Jianpeng Cheng, Li Dong, and Mirella Lapata. 2016. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733* (2016).

[21] Juliet Corbin and Anselm Strauss. 2014. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications.

[22] Anthony Desnos et al. 2013. Androguard-reverse engineering, malware and goodware analysis of Android applications. *URL code. google. com/p/androguard* 153 (2013).

[23] Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608* (2017).

[24] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 486–497.

[25] Lingling Fan, Minhui Xue, Sen Chen, Lihua Xu, and Haojin Zhu. 2016. Poster: Accuracy vs. time cost: Detecting Android malware through pareto ensemble pruning. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 1748–1750.

[26] Ruitao Feng, Sen Chen, Meng Guozhu Xie, Xiaofei, and Liu Yang Shang, Wei-Lin. 2020. A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices. *IEEE Transactions on Information Forensics and Security* (2020).

[27] Ruitao Feng, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, and Shang-Wei Lin. 2019. Mobidroid: A performance-sensitive malware detection system on mobile platform. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 61–70.

[28] Ruitao Feng, Jing Qiang Lim, Sen Chen, Shang-Wei Lin, and Yang Liu. 2020. SeqMobile: An Efficient Sequence-Based Malware Detection System Using RNN on Mobile Devices. In *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE.

[29] Hossein Fereidooni, Mauro Conti, Danfeng Yao, and Alessandro Sperduti. 2016. ANASTASIA: Android mAlware detection using STatic analySIs of Applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5.

[30] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 3 (2018), 1–29.

[31] Hamidreza Ghader and Christof Monz. 2017. What does Attention in Neural Machine Translation Pay Attention to?. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017 - Volume 1: Long Papers*. 30–39.

[32] Google. 2019. Documentation for app developers. *developer.android.google.cn* (2019). https://developer.android.google.cn/docs

[33] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[34] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. 2015. Needles in a haystack: mining information from public dynamic analysis sandboxes for malware intelligence. In *24th USENIX Security Symposium (USENIX Security 15)*. 1057–1072.

[35] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2019. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)* 51, 5 (2019), 93.

[36] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 364–379.

[37] Weiwei Hu and Ying Tan. 2017. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983* (2017).

[38] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. 2018. A multimodal deep learning method for Android malware detection using various features. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018),

773–788.

[39] Josua Krause, Adam Perer, and Enrico Bertini. 2014. INFUSE: interactive feature selection for predictive modeling of high dimensional data. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 1614–1623.

[40] Yusi Lei, Sen Chen, Lingling Fan, Fu Song, and Yang Liu. 2020. Advanced Evasion Attacks and Mitigations on Practical ML-Based Phishing Website Classifiers. *arXiv preprint arXiv:2004.06954* (2020).

[41] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-an, and Heng Ye. 2018. Significant permission identification for machine-learning-based Android malware detection. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3216–3225.

[42] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering*, Vol. 1. IEEE Press, 280–291.

[43] Cullen Linn and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 290–299.

[44] Zachary C Lipton. 2016. The mythos of model interpretability. *arXiv preprint arXiv:1606.03490* (2016).

[45] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Advances in neural information processing systems*. 4765–4774.

[46] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[47] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. 2017. Deep Android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 301–308.

[48] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. 2018. Explaining black-box Android malware detection. In *2018 26th European Signal Processing Conference (EUSIPCO)*. IEEE, 524–528.

[49] Tim Miller. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence* 267 (2019), 1–38.

[50] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. 2018. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing* 73 (2018), 1–15.

[51] Naser Peiravian and Xingquan Zhu. 2013. Machine learning for Android malware detection using permission and api calls. In *2013 IEEE 25th international conference on tools with artificial intelligence*. IEEE, 300–305.

[52] Brett Poulin, Roman Eisner, Duane Szafron, Paul Lu, Russell Greiner, David S Wishart, Alona Fyshe, Brandon Pearcy, Cam MacDonell, and John Anvik. 2006. Visual explanation of evidence with additive classifiers. In *Proceedings of the National Conference on Artificial Intelligence*, Vol. 21. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 1822.

[53] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. Droidchameleon: evaluating Android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 329–334.

[54] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why should I trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 1135–1144.

[55] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. 2017. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296* (2017).

[56] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*. 120–131.

[57] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[58] Chongbin Tang, Sen Chen, Lingling Fan, Lihua Xu, Yang Liu, Zhushou Tang, and Liang Dou. 2019. A large-scale empirical study on industrial fake apps. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 183–192.

[59] Stef Van Den Elzen and Jarke J van Wijk. 2011. Baobabview: Interactive construction and analysis of decision trees. In *2011 IEEE conference on visual analytics science and technology (VAST)*. IEEE, 151–160.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[61] Lilian Weng. 2018. Attention? Attention! *lilianweng.github.io/lil-log* (2018). http://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

[62] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. 2014. AirBag: Boosting Smartphone Resistance to Malware Infection. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[63] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*. IEEE, 62–69.

[64] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. 2016. Effective detection of Android malware based on the usage of data flow APIs and machine learning. *Information and Software Technology* 75 (2016), 17–25.

[65] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*. 2048–2057.

[66] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 2048–2057.

[67] Ke Xu, Yingjiu Li, Robert H Deng, and Kai Chen. 2018. DeepRefiner: Multi-layer Android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 473–487.

[68] Lok Kwong Yan and Heng Yin. 2012. Droidscope: seamlessly reconstructing the os and Dalvik semantic views for dynamic Android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 569–584.

[69] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*. 1480–1489.

[70] Suleiman Y Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. 2013. A new Android malware detection approach using bayesian classification. In *2013 IEEE 27th international conference on advanced information networking and applications (AINA)*. IEEE, 121–128.

[71] Wei Yu, Linqiang Ge, Guobin Xu, and Xinwen Fu. 2014. Towards neural network based malware detection on Android mobile devices. In *Cybersecurity Systems for Human Cognition Augmentation*. Springer, 99–117.

[72] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. 2016. Droiddetector: Android malware characterization and detection using deep learning. *Tsinghua Science and Technology* 21, 1 (2016), 114–123.

[73] Bolei Zhou, Yiyou Sun, David Bau, and Antonio Torralba. 2018. Interpretable basis decomposition for visual explanation. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 119–134.

[74] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM Conference on Data and Application Security and Privacy*. ACM, 185–196.

[75] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.

[76] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets.. In *NDSS*, Vol. 25. 50–52.