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

BOZHI WU, Nanyang Technological University, Singapore
SEN CHEN, Nanyang Technological University, Singapore
CUIYUN GAO, Harbin Institute of Technology (Shenzhen), China
LINGLING FAN, Nanyang Technological University, Singapore
YANG LIU, Nanyang Technological University, Singapore
WEIPING WEN, Peking University, China
MICHAEL 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 comparing 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.

Additional Key Words and Phrases: Malware Classification, Interpretability, Machine Learning

ACM Reference Format:
Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael Lyu. 2020. Why an Android App is Classified as Malware? Towards Malware Classification Interpretation. 1, 1 (April 2020), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [13, 14], consequently, hackers shift their attention on mobile devices and try to perform malicious behaviors through Android

Sen Chen is the corresponding author. Email: ecnuchensen@gmail.com.
Authors' addresses: Bozhi Wu, wubozhi@pku.edu.cn, Nanyang Technological University, Singapore; Sen Chen, chensen@ntu.edu.sg, Nanyang Technological University, Singapore; Cuiyun Gao, cygao@cse.cuhk.edu.hk, Harbin Institute of Technology (Shenzhen), Shenzhen, China; Lingling Fan, llfan@ntu.edu.sg, Nanyang Technological University, Singapore; Yang Liu, yangliu@ntu.edu.sg, Nanyang Technological University, Singapore; Weiping Wen, weipingwen@pku.edu.cn, Peking University, Bei Jing, China; Michael Lyu, lyu@cse.cuhk.edu.hk, Chinese University of Hong Kong, Hong Kong, China.

2020. XXXX-XXXX/2020/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

apps. It is not surprising that a number of approaches have been proposed for detecting Android malware. Specifically, traditional signature-based approaches [46, 64, 66] require frequent updates of the signature database and fail to be effective in detecting emerging malware. Behavior-based approaches [28, 47, 52, 58] also rely on the predefined malicious behaviors, which is limited by the analysis of existing malicious samples. Data flow-based approaches [8, 27, 36] 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 [5], SVM [6], Random Forest [44], and XGboost [24]) to classify and categorize malware. In these approaches, Android permissions and API calls are the commonly-used feature types [6, 15, 17, 53], 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 [23, 32, 62] 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 reliance 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 [11, 15, 16, 18, 31, 34]. 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 [6] 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 [45], 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 [7, 25, 56, 63]. 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. Because 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 97.04% accuracy, and can identify the malicious behaviors that are validated by cross validation manually. In addition, we compare the results with the state-of-the-art 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 comprehensive manual verification of the capability of XMal regarding interpretability, and also provide a comparison study with the state-of-the-arts to demonstrate the effectiveness of XMal.
- We present a case study 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. [21], 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 [45] and LEMNA [30]. On the other head, researchers have also done some studies in areas of text categorization and image classification. For example, Arras et al. [7] 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. [63] 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 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 [6] 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.

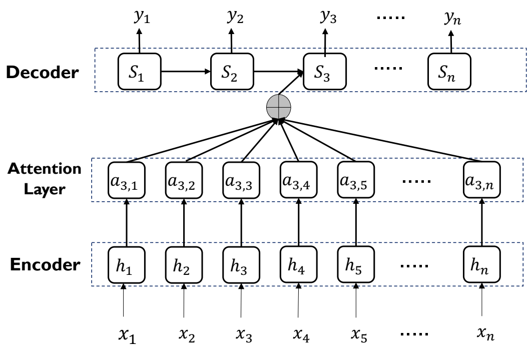
Decoder
Attention Layer
Encoder
y_1 y_2 y_3 ····· y_n
S_1 S_2 S_3 ····· S_n
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$ ····· $a_{3,n}$
h_1 h_2 h_3 h_4 h_5 ····· h_n
x_1 x_2 x_3 x_4 x_5 ····· x_n

Fig. 1. Attention in machine translation

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. [41] 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 [51]. 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. [9] 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 information, making the model's predictions more accurate, and make the prediction more understandable. For example, Xu et al. [55] 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 [45], 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

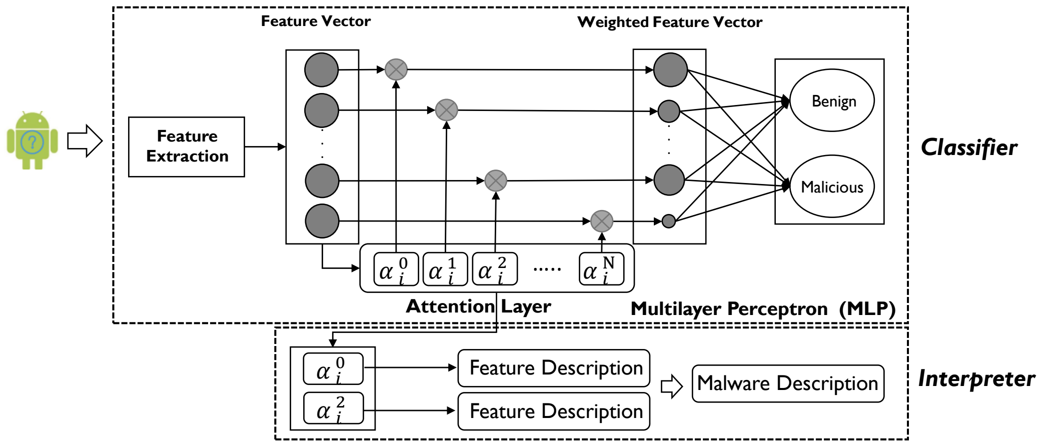

Fig. 2. Overview of our approach (XMal)

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 [26]. The details of each component are elaborated on 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 [42]. A lot of studies used these two features as significant features for classifying Android malware, such as Drebin [6], DriodAPIMiner [5], DroidMat [53] and many other previous studies [15, 17, 24, 32, 40, 57, 60, 62]. 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. [35] 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 [15] and select 158 features (including 97 API calls and 61 permissions) for our study by using manual statistical pruning method in [15] 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. In order to extract API calls and permissions, we utilize ANDROGUARD [20] to extract API calls and permissions from APK file, which are used to construct the feature vector. Here we denote a sample set by $\{(\boldsymbol{x_i}, y_i) \in (X, Y)\}_{i=1}^{M}$, where $\boldsymbol{x_i} = (x_i^{(1)}, x_i^{(2)}, x_i^{(3)}, ..., x_i^{(N)})$ 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:

$$\boldsymbol{e_{ij}} = score(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j), \tag{1}$$

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 [39] 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.

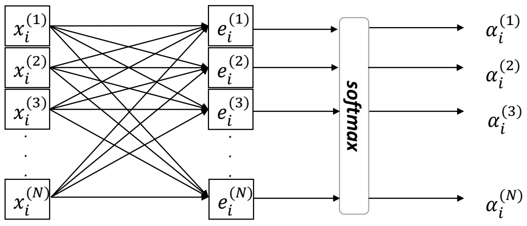

Fig. 3. Attention layer in XMal

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 super parameter. 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 [45]. The number for *n* is a heuristic value depending on concrete scenarios. According to the experiments on our dataset, 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, which makes 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, meanwhile, the deep learning models still cannot be interpreted accurately. How to interpret deep neural networks is an open challenge so far, which is also belonging to our future work.

3.3 Malware Description Generation

In order to generate malicious behavior description for malware, we first match the malware features to their semantics. As mentioned before, we select a 158-dimensional feature vector as input to train the classifiers. We search Android developer documentation [26] for the semantics of each feature according to their names. As we know, the Android developer documentation has a detailed functional description for each API and permission. We download the detailed functional description of each feature, then use the feature and the corresponding functional description to build a database. But the functional descriptions have too many details and are difficult to understand all of them. Therefore, we simplify and generalize them into simple semantics. For example, the functional description of URL.openConnection is “Returns a URLConnection instance that represents a connection to the remote object referred to by the URL”. We generalize it as “Access the Internet". Similarly, TelephonyManager.getSubscriberId is generalized as “Collect IMSI". 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 can be combined into one semantic. For example, TelephonyManager.getDeviceId and TelephonyManager.getSubscriberId are both about information collection and can be combined into “Collect IMEI/IMSI”. 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 malware, We define some ordering rules to arrange the semantics. 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”.

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, where s_i is the i-th semantics. According to Rule 1, if key features belong to a same functionality, they are assigned 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 combined it with s and then update s in S. After that, we convert the semantics into descriptions one by one.

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. In addition, in order to demonstrate the superiority of the proposed method, we compare it with the state-of-the-art and conduct a quantitative analysis. Finally, we present a case study and provide an in-depth discussion to help to understand our approach.

4.1 Experimental Setup

4.1.1 Dataset. To investigate the malware detection accuracy and interpretability of the proposed method XMal, we first collect a large amount of malware from two sources: 10,010 samples from the National Internet Emergency Center and 5,560 samples from Drebin [6]. Most of the samples from the National Internet Emergency Center are the recent malicious samples rather than the old dataset such as Gemome [65] in 2011. These malware samples include all varieties of threats for Android, such as data leakage, phishing, trojans, spyware, and root exploits. Apart from these malicious apps, we also collected 20,120 benign apps from Google Play Store, and removed the ones that are classified as malware by VirusTotal service [4].

4.1.2 Evaluation Design. To evaluate the effectiveness of our proposed approach to interpreting malware, according to the malware family tags provided to 5,560 malware samples by Drebin [6], we select the top 16 malware families with the largest number of samples (cf. Table 1). Since some malware families have too few samples to validate the interpretable results. To finish the following manual cross-validation of the explanation results of malware samples, we randomly select 10 samples from each malware family (i.e., 160 samples in total) to form a malware test and collect the corresponding expert analysis of each family from Symantec [3] and Microsoft [2]. We also randomly select 10 samples and obtain the corresponding expert analysis from National Internet Emergency Center for malware test. We utilize XMal to generate the malicious behavior descriptions of the 170 test samples and evaluate how well they match with the expert analysis. In addition, to further evaluate whether XMal can explain why the application is misclassified as benign or malicious, we randomly select 170 benign apps as the benign test set to conduct the experiments. We manually analyze the samples that are misclassified and explain why the misclassification occurred.

To evaluate the detection accuracy of XMal, we randomly split the 15,400 malware samples and 19,950 benign apps into a training set (70%) and a test set (30%). Meanwhile, we extract 158-dimensional feature vectors (including 97 API calls and 61 permissions) from the training set and the test set and use them to train and test XMal. In order to check whether the classification accuracy of the test samples in the interpretability experiment is sufficiently high, we also use XMal to test

Table 1. Top 16 malware families

Adrd	BaseBridge	DroidDream	SendPay
DroidKungFu	Geinimi	GinMaster	Iconosys
FakeDoc	Plankton	FakeInstaller	SMSreg
Gappusin	Kmin	MobileTx	Opfake

the 170 malware samples and the 170 benign samples. After that, we conduct a comprehensive manual verification of the capability of XMal regarding interpretability.

To demonstrate the effectiveness of XMal compared with the state-of-the-arts, we compare XMal with Drebin [6] and LIME [45]. Drebin is from Android malware research field, and LIME proposes a model-agnostic method to interpret individual model prediction. In order to conduct comparison experiments, we first re-implemented these two models based on their published research papers. 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. 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 using manual statistical pruning used in [15]. We further obtain the weight of features by acquiring the coefficient of the liner SVM. The features with the highest weights are used to interpret the classification result. For LIME, since it is a model-agnostic method, we perform it on the MLP model. Specifically, we extract the same features as XMal from the data set mentioned in the accuracy experiment (i.e., the extracted features include 97 API calls and 61 permissions) to train and test the MLP model. Then we perform LIME on the MLP model to obtain the feature weight of each test sample. The feature with the highest weight is used to explain why an app is classified as malware or benign. We use the 170 malware test samples to compare the interpretability results of Drebin, LIME and XMal based on how well the features with maximum weights can match the behaviors of the corresponding malware family.

4.2 Evaluation on Detection Accuracy

The accuracy of the three models are all above 95%, while XMal achieves 97.04% detection accuracy, outperforms the other two methods as shown in Table 2. Therefore, the detection accuracy of the three methods is high enough to ensure that the interpretability analysis is meaningful. 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 the next section, we will analyze why those samples are misclassified.

4.3 Evaluation on Interpretability

We employ XMal on the 170 malware and 170 benign apps, and obtain the corresponding explanations of the classification results. According to our experimental observation and cross-validation phase, we find it appropriate to choose 6 for the super parameter n in our experiments, since 6 key

Table 2. Detection results of three models

Models	**Drebin**	**MLP in LIME**	**XMal**
Recall	94.90%	97.13%	98.24%
Precision	95.94%	96.38%	96.33%
Accuracy	95.24%	96.50%	97.04%

Table 3. Detection accuracy of the top 16 malware families

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%

features can cover the key malicious behaviors in the 16 malware families. Note that, the number of key features may be fewer than 6, because some samples have < 6 features in total. 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 the classification. The interpretability results are as shown in Table 4. 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 4, 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 4 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 4, 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,

Table 4. 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. Re-execute itself when the mobile device is booted up. 2. 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
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	Send the premium SMS
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.

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 4, 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 4. XMal provides a fairly reasonable explanation for the classification results. However, there are also some exceptions, such as a sample of FakeInstaller shown in Table 4. 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

Table 5. 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

contacts. XMal captures the malicious behavior of collecting contact information but mistakes 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 4.2, 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 5 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.

4.4 Comparison with the State-of-the-arts

In this section, to demonstrate the interpreting effectiveness of our XMal, we compare it with two state-of-the-art interpretable ML systems, Drebin [6] and LIME [45], on three representative malware families (i.e., Adrd, GinMaster, and MobileTx). In order to compare the three methods, we first calculate all key features for all samples in the 16 families. We choose 6 key features in this comparison experiment. As shown in Table 6, we select one sample from each family, and obtain the key features extracted by the three methods, respectively. We compare the interpretability of these three methods based on how well the key features match the expert analysis of the corresponding malware family, and discuss the performance of the three methods based on the key features of the three malware families generated by them. In fact, the feature itself contains semantic to some extent, from which we can probably know the relation between the feature and the malicious behavior. For example, malware attempting to launch with system startup require to apply for the permission, RECEIVE_BOOT_COMPLETED; malware attempting to steal information require to apply for the permission, READ_PHONE_STATE, and may need to execute TelephonyManager.getSubscriberId and TelephonyManager.getDeviceId; and malware attempting to access the internet may require to execute URL.openConnection.

As shown in Table 6, 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 6. 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 6. Comparison of three approaches (i.e., Drebin, LIME, and XMal). The bold texts refer to key features that cannot match the real malicious behavior. The texts on a gray background refer to the expert analysis corresponding to each sample.

	Key Features		
	Drebin	LIME	XMal
Adrd	**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
	1. Re-execute itself when the mobile device is booted up. 2. Steal some info and send to remote server		
	Drebin	LIME	XMal
GinMaster	**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**	
	1. Steal info from the device. 2. Send info to remote server. 3. The malicious service is triggered when the device finishes a boot.		
	Drebin	LIME	XMal
MobileTx	**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**	
	1. Steal info from the compromised device. 2. Send SMS messages to premium-rate number.		
	Drebin	LIME	XMal
blackgame	**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
	1. Send SMS message to premium-rate num. 2. Obtain phone num and device info and upload it to the remote server.		
	Drebin	LIME	XMal
xunbaikew1	**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**	
	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 these three methods in Fig. 4. 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. The similar phenomenon occurs on LIME for the feature NotificationManager.notify.

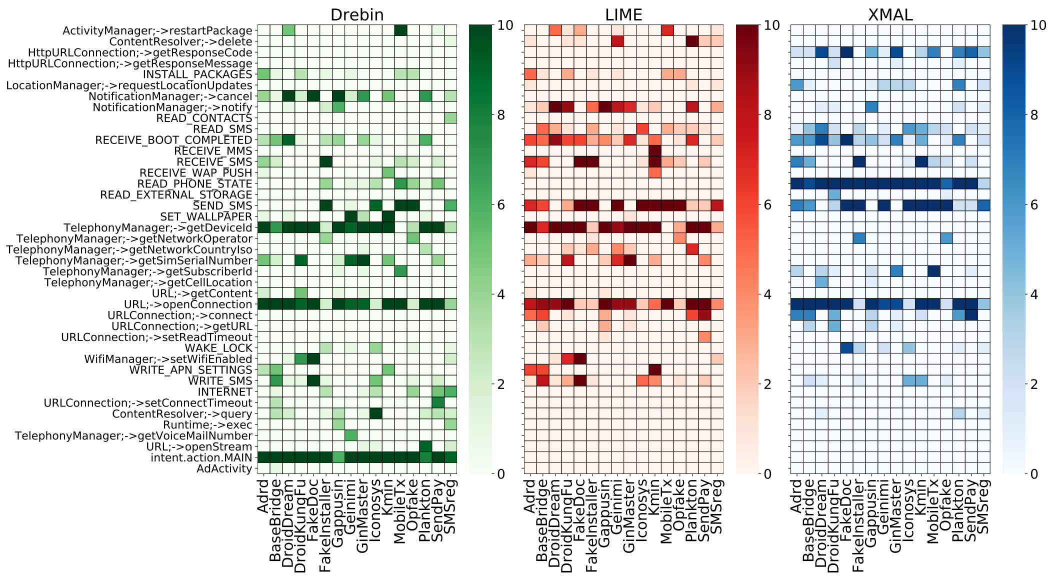

Fig. 4. Total number of key features across all malware samples generated by the Drebin, LIME, and XMal

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 6). 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 limitations of XMal and summarize open challenges in the interpretability of Android malware detection according to our study.

5.1 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 [32] 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 re-execute 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. But 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.

5.2 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 4.2, 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 [37] 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 [12, 48], machine learning methods have become very popular and have achieved great success in Android malware [5, 6, 15, 17, 18, 22, 24, 35, 44, 54, 60]. For example, Aafer et al. [5] 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. [60] presented a method to detect Android malware based on Bayesian Classification models obtained from API calls, system commands and permissions. Wu et al. [54] 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. [35] 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 [6], Random forest [44], and XGboost [24] 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 [23, 32, 40, 57, 61]. Yu et al. [61] proposed to train a malware detection model by using a representative machine learning technique, called ANN. McLaughlin et al. [40] 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. [32] 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. [57] 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 dimension reduction, clustering, classification and regression analysis. Elzen et al. [49] 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. [43] 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. [33] 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. [45] 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. [30] 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 [29, 38] 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. [9] first proposed to solve the problem of incapability of remembering long source sentences in neural machine translation (NMT). Xu et al. [55] 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. [50] 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 [19], video summarization [10] and document classification [59]. 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.

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] 2019. *Microsoft.* https://www.symantec.com/https://www.microsoft.com/en-us/wdsi/threats/
[3] 2019. *Symantec.* https://www.symantec.com/
[4] 2019. *Virustotal.* https://www.virustotal.com/gui/home/upload
[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[10] 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).

[11] 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).

[12] 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).

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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)*.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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).

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. 2011. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones.. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, Vol. 11. 17–33.

[47] 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)*.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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)*.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

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

[66] 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.