A Knowledge Graph-based Sensitive Feature Selection for Android Malware Classification

Duoyuan Ma*, Yude Bai*, Zhenchang Xing†, Lintan Sun‡ and Xiaohong Li*
*Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University, Tianjin, China
†Research School of Computer Science, Australian National University, Data61 CSIRO, Australia
‡State Grid Customer Service Center, Tianjin, China
{2018216033,baiyude,xiaohongli}@tju.edu.cn, zhenchang.xing@anu.edu.au, dzsltan@163.com

2020 27th Asia-Pacific Software Engineering Conference (APSEC) | 978-1-7281-9553-7/20/$31.00 ©2020 IEEE | DOI: 10.1109/APSEC51365.2020.00027

***Abstract*—The rapid increase in Android malware has brought great challenges to malware analysis. To deal with such a severe situation, it has been proposed an effective way which groups malware with common behaviors into the same malware family. Although there are many methods for malware family classification, the most critical and primary step is always the definition of sensitive behavior in an application, which will be beneficial for the later classification task. Much existing literature has manually selected sensitive features, such as permission, or even designed graph-based features via the control flow graph. They heavily depend on expert knowledge and time-consuming malware application analysis, which means it has to focus on the malware itself to dig out valuable security knowledge at first. However, the zooming malware overwhelms such expensive feature definition methods. To overcome such a problem, we adopt a knowledge graph-based sensitive feature selection method for Android malware classification. Based on the Android Developer documentation, an Android API knowledge graph is constructed at first. We can obtain not only permission but also related critical API from this graph. Note that both hyperlink relation and similarity relation are used to find out the critical API. With the knowledge graph-based sensitive features, we represent each Android malware as a boolean feature vector and send it in to a machine learning classifier for malware classification. We evaluate our proposed methods on three well-known Android malware datasets, such as Genome, Drebin, and AMD. The experimental results show that: 1) our proposed sensitive API is advantageous for malware detection; 2) API chosen by similarity relation can marginally improve performance; 3) different permission groups also make an influence for classification.**

***Index Terms*—Android malware family, Knowledge graph, Machine learning**

I. Introduction

As the most popular mobile operating system, Android's smartphone share hovers around 86.1% in 2019 [14]. Simultaneously, Android smartphones are the most common target for malware, which is twenty times more than other smartphones. A recent threat report from McAfee Labs shows that an average of 2 million new mobile malware was captured quarterly throughout 2017 [21]. However, the analysis of new malware needs plenty of time [38]. As a result, malware analysis is increasingly overwhelmed by the number of newly discovered malware that needs to be analyzed.

To solve the problem, training a suitable and efficient classifier to classify the malware into an existing malware family is often a common and effective method. When training a classifier, it is inevitable to embed malware into the sample space for vectorization. Therefore, an important question is choosing what kind of features for malware vectorization. Extensive syntactic features have been used in this classification task, such as Android permissions [32], critical API calls [10]. However, using such syntactic features often requires a lot of expert experience, which costs much time and effort. Meanwhile, with the continuous change of Android versions, expert screening efficiency is also very low. After our preliminary statistics, the official Android documentation contains about 80000 APIs. It is not practical to manually filter these APIs. An automatic and accurate method is urgently needed. Statistical-based feature selection methods often have a strong dependence on the data set, which causes it to not adapt to the Android version update speed and the speed of malware growth.

In an actual situation, some permissions and APIs are not only used by malicious software, but benign software often applies to these. Therefore, it is not easy to classify malware from a single feature effectively. However, we can analyze the combined use of multiple sensitive permissions and their related APIs to classify malware through a set of sensitive APIs instead of a single API. It can also be seen that the importance of screening appropriate permissions and APIs.

To find an useful permission-API set, it is good to start with the API documentation. The API documentation contains detailed information on the API usage process, such as functions, parameters, return values, conditions of use, exceptions are thrown, etc. This information has a guiding role for developers in the development process and helps researchers analyze malware behavior and the occurrence of vulnerabilities. With this information's help, we can obtain the specific content of the permission and the function of the API to determine its relationship with the malware.

After observing the official API documentation, we found that permissions are divided into four categories, which are Normal, Signature, Dangerous, SignatureOrSystem, and not all permissions are related to user privacy (sensitive operations). Among them, the official description of dangerous permission is: "higher-risk permission that would give requesting application access to private user data or control over the device

2640-0715/20/$31.00 ©2020 IEEE
DOI 10.1109/APSEC51365.2020.00027

TABLE I
API EXAMPLE.

API name	android.telecom.TelecomManager# getCallCapablePhoneAccounts()
related permission name	android.Manifest.permission# READ_PHONE_STATE
API description	Returns a list of PhoneAccountHandles which can be used to make and receive phone calls. The returned list includes only those accounts which have been explicitly enabled by the user. Requires permission: Manifest.permission.READ_PHONE_STATE

that can negatively impact the user." [15] From this, we can see that the API which requires dangerous permission should be a sensitive API involving risky operations such as getting private user data and device control.

Besides, we collected a total of 79290 APIs from the Android API documentation. Initial observations found that APIs related to sensitive operations such as reading SMS, obtaining account information, and other user privacy information may be less than 10%. This observation indicates that among these large number of APIs and Permissions, something can help the classification task and some that do not work or even counteract the classification task. It is not easy to find such an API directly from the official documentation. Because this information comes from web pages, the content is inconvenient to obtain, and the structure is messy. The messy data is not conducive to the practical analysis of the data. The concept of a knowledge graph can solve this problem well. The borrowing knowledge graph can efficiently organize unstructured data into structured information, and through relational reasoning, more useful additional knowledge can be discovered. The API knowledge graph may allow us to discover more potential relationships and knowledge of APIs and permissions.

We can transfer the analysis of malware to the analysis of the API knowledge graph. Malware will proliferate and change constantly, but the graph tends to be stable. When a new API or API function changes, we only need to modify a small part of the graph's nodes and relationships. The overall graph does not Great changes have taken place, so the cost of manual work can be greatly reduced.

In this paper, we also use the knowledge graph to structure API document information and establish the relationship between APIs and APIs and APIs and Permissions. Discover the APIs associated with dangerous permission through the graph TableI shows an example of API require permission and observe the effectiveness of these APIs for malware classification tasks. We also try to compare the semantic similarity of their functional description parts to determine whether these APIs have sensitive operations for some special APIs. We also need to test their performance on classification tasks.

Specifically, our experiments answer the following four research questions:

- RQ1: Is it appropriate to use APIs directly related to permission whose protection-level is dangerous?
- RQ2: How many APIs that are indirectly related to permission whose protection-level is dangerous is sufficient for classification tasks?
- RQ3: Is it possible to discover hidden sensitive APIs that cannot be obtained by searching the API knowledge graph?
- RQ4: Which kind of API group is more helpful for malware classification?

To answer these four research questions, we use three datasets of Android applications. The first dataset is the well-known Android Malware Genome Project [39] (consisting of 1260 malware in 49 families). The second dataset is the Drebin dataset [1] (consisting of 5560 malware in 179 families) using the malware data crawled from VirusShare. The last dataset is the AMD dataset [33](consisting of 24553 malware in 72 families).

After reverse engineering, all applications are decompiled and represented as a primary syntactic feature vector, which is the classifier's input. Then classify and predict it and analyze the results.

In summary, our work contributes to the following:

- We obtained API and other information from the official Android API documentation and constructed the API knowledge graph based on the data structure characteristics and our classification tasks.
- By constructing a knowledge graph, we propose a method for automatically screening out a list of candidate APIs capable of performing classification tasks based on knowledge.

II. RLATED WORK

A. Malware Detection

As mentioned earlier, Android malware detection and classification fall into two general categories: 1) signature-based, 2) machine learning-based.

Signature-based approaches [10], [12], [27], [13] try to identify specific functional mode in Android source code for malware analysis. By creating API call graphs for malware applications, DroidLegacy [10] provides a way to identify and classify piggybacked malware applications into corresponding families. Apposcopy [12] combines static taint analysis and call graphs among components to establish semantic attributes and then recognizes Android applications based on ICCG. Dendroid [27] uses a text mining method (based on TF-IDF) to analyze the code structure of the application through CFG (Control Flow Graph). Astroid [13] tries to look for a maximally suspicious common subgraph for each malware family and then uses a combination of static analysis and approximate signature matching algorithm to analyze which family does the Android application belong to.

Learning-based approaches [3], [11], [36] aim to automatically extract some significant features as the representation of applications and then use a machine learning method that is training the classifier for classification. MudFlow [3] detects

sets of malware behaviors through data flow (the source-and-sink approach of FlowDroid [2]) by v-SVM, which is a binary classification to determine whether an application is a malware or not. FalDroid [11] designs Fregraphs, a novel graph-based feature used to represent the shared behavior of Android malware families. Then it applies a machine learning algorithm, such as SVM and KNN (k-nearest neighbor), for training diverse classifiers. DroidSIFT [36] also adopts a graph-based method. It can identify Android malware with a transformation attack by extracting a weighted contextual API dependency graph to construct a semantic feature set.

However, there are shortcomings in both signature-based approaches and learning-based approaches. They require trained security experts to manually select the appropriate function for signature-based methods, especially when defining the function from a constructed graphical mode. [13]. Meanwhile, learning-based approaches produce results that are difficult for security experts to interpret [36]. The right way is to combine the two, using the API and permission mentioned in the signature-based method as features, and then use the learning-based method for training. So our classification experiment is based on this combined method, using API and Permission as the characteristics of the software, and then training the classifier to classify. Zhang et al. [37]proposed a method to classifier Android malware with weaker tags, which integrates heterogeneous information from multiple sources, including the results of static code analysis, meta-information of Android applications, and original tags of antivirus engines.

B. API Documentation and Knowledge Graph

There are many researchs about the quality of API documentation [30], [18], [24], [25], [40], and they study the lack, incompleteness, and obsolescence of API documentation. Many techniques have been proposed to solve these problems, for example, the automatic generation and continuous updating of API documents [28], [22], [17]. In addition to quality, traceability research of API documentation has also received much attention. For example, Bacchelli et al. [5], [4] developed Miler for API extraction and link infrastructure. Dagenaisd et al. [9] developed RecoDoc to extract Java APIs from multiple sources and then perform traceable link recovery between different sources. Subramanian et al. [28] link API references in some code fragment to APIs in the knowledge base by analyzing code context information.

In addition to the research on API documents' quality and traceability, the research on knowledge mining of API documents has also made some progress. Li et al. [20] build an API caveat knowledge graph to improve the accessibility of API caveats. Sun et al. [29] integrate the API knowledge graph in IDE to prompt developers with API usage examples. These works give us the idea of constructing a knowledge graph to improve API and Permission connectivity.

III. METHODOLOGY

This article aims to design a method based on a knowledge graph for API Feature Selection, and in this section, we define our research problem, introduce the API Knowledge Graph and how to select sensitive API, and present the machine learning methods we use for malware family classification.

A. API Knowledge Graph

Fig. 1 shows the steps in our approach: mining API knowledge graph from API documentation and searching candidate sensitive APIs based on the mined knowledge graph. The work contains three steps: preprocess API documentation from the website, build an API Knowledge graph, select candidate sensitive APIs from the graph.

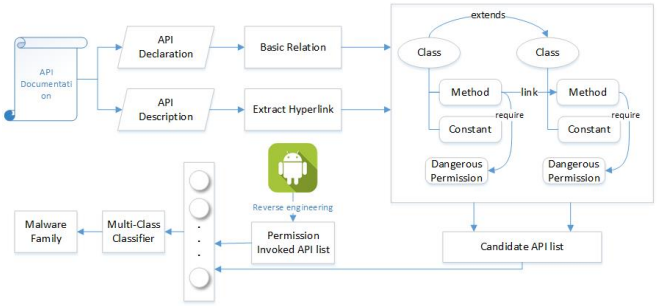

Fig. 1. The overview of framework.

1) Data and Preprocess: API reference documentation provides a detailed message about declarations, functions, and package, class, and API conditions. This information is an essential material for both developers and researchers. So, firstly we have to get these data.

We get Android API reference documentation from the website using BeautifulSoup[1]. We consider one website page as a class-documentation that contains three parts (class and its description, constants declared by this class and their description, APIs declared by this class and their description). Through the structural characteristics of the web page, we remove the web page elements like <code>, <image>, and <script> except those three parts.

The description parts also need further natural language processing. For API, because its tokens always out of natural vocabulary like ".", "()", "[]", "_", "#", it's difficult to identify API through traditional NLP tools.For example: an API description "boolean registerAntennaInfoListener(Executor executor, Listener listener) Registers a Gnss Antenna Info listener. Only expect results if GnssCapabilities#hasGnssAntennaInfo() shows that antenna info is supported. Requires Manifest.permission.ACCESS_FINE_LOCATION" which contains an API name "GnssCapabilities#hasGnssAntennaInfo()" and a constant "Manifest.permission.ACCESS_FINE_LOCATION". If we use traditional natural language tools to process directly, these special tokens' structure will be destroyed. Although it is not easy to identify them by using common NLP tools but for java API, we can identify API tokens by using Regular expression because of the special structure named Hump nomenclature.

[1]https: (www.crummy .com/software/BeautifulSoup/

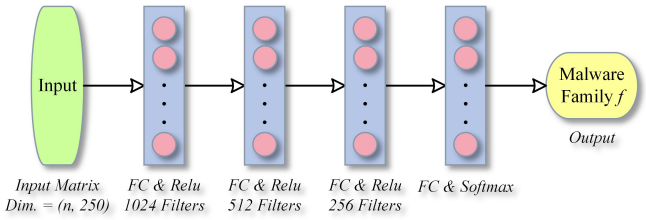

Fig. 2. The overflow of MLP.

After identifying these API documents' unique tokens, the remaining parts of the document can use the existing NLP tools such as nltk[2], Stanford CoreNLP[3], etc. to complete the natural language processing of clauses and words.

2) API Knowledge Graph Construction: API reference documentation gives the details of API. So we can use this message to extract basic API entities and relations between API entities. In this work, we consider classes, methods, and constants as entities (for parameters and return types; we consider them as properties of methods to make the graph concise). Relations contain declare, extends, implements, parameter primitive-type, return primitive-type, and throw an exception. Either API entity or constant entity in the graph is identified by its full name(to distinguish APIs with different libraries with the same name) with its parameter list(to distinguish overloaded functions of the same name).

After that, we describe all APIs and build a "Hyperlink" relationship for each entity based on the hyperlinks in the description to ensure API accessibility between different classes.

B. Classification task

1) Searching for APIs in the Knowledge Graph: By observing the API documentation, we found that Android permission has four kinds of protection-level: normal, signature,signatureorsystem, dangerous [15]. The permission whose protection_level is dangerous(30 in total) should get attention first. TableII shows 11 permission group classified by officially [16]. Moreover, these 30 dangerous permissions can be divided into 11 groups according to their function. As can be seen from the functional description of the permission group, dangerous permission is closely related to mobile phone users' privacy information.

Therefore, APIs related to dangerous permission will also involve the private information of users. In other words, these APIs related to dangerous-permission should be classified as sensitive APIs. Compared with other APIs not related to dangerous permission, these APIs can better characterize malware and can be more suitable as features of Android malware classifiers. For example, given a dangerous permission READ_PHONE_STATE from permission group PHONE, We search all APIs that directly require this permission through the knowledge graph. After that, we search for other APIs related to these permissions.

[2]https://github.com/nltk/nltk
[3]https://stanfordnlp.github.io/CoreNLP

TABLE II
PERMISSION GROUP AND ITS FUNCTION.

GroupName	Function
ACTIVITY_RECOGNITION	Used for permissions that are associated with activity recognition.
CALENDAR	Used for runtime permissions related to user's calendar.
CALL_LOG	Used for permissions that are associated telephony features.
CAMERA	Used for permissions that are associated with accessing camera or capturing images/video from the device.
CONTACTS	Used for runtime permissions related to contacts and profiles on this device.
LOCATION	Used for permissions that allow accessing the device location.
MICROPHONE	Used for permissions that are associated with accessing microphone audio from the device.
PHONE	Used for permissions that are associated telephony features.
SENSORS	Used for permissions that are associated with accessing body or environmental sensors.
SMS	Used for runtime permissions related to user's SMS messages.
STORAGE	Used for runtime permissions related to the shared external storage.

2) Classifier: Multi-Layer Perceptron (MLP): To verify the API list's validity through the API knowledge graph as a feature, we conducted classification tests on three malware data sets. The results of the classification reflect the effectiveness of the selected APIs as features.

We adopts the Multi-layer Perceptron (i.e. MLP) [19] for multi-class malware family classification, which is a feedforward artificial neural network [8]. We construct three different fully connected layers as the hidden layer to learn features from the input vector (see Fig. 2). Then, the last hidden layer's output is passed into the final classifier layer with a softmax activation function [6], which generates the probability that a malware application belongs to the fth family. The family with the highest probability is regarded as the family label of the given malware application.

C. Discover hidden APIs

We call APIs that are directly or indirectly related to user privacy information as sensitive APIs. We can get these APIs by searching for nodes related to dangerous permission in the API knowledge graph. However, not all APIs can establish hyperlink relationships through descriptions in the official documentation, and even a few APIs may not contain descriptions. For those APIs that do not have descriptions, we will not consider them here. For APIs that contain descriptions but can not establish hyperlinks with other permissions or APIs, some NLP methods can be used to explore some of the APIs related to dangerous permissions. For example:"String getMessageBody() Returns the message body as a String, if it exists and is text based.". This API has neither a description of

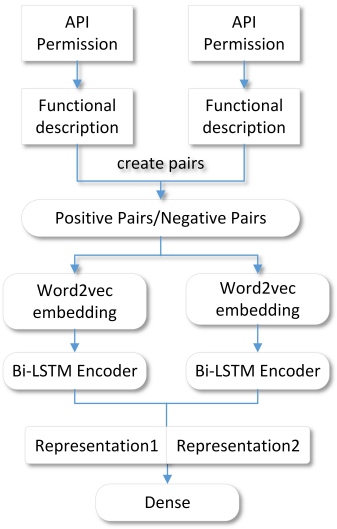

Fig. 3. The overview of Siamese-BiLSTM text similarity calculation.

requiring permission nor a description related to other APIs. However, as can be seen from his description, this api is related to dangerous permission READ_SMS whose description is "Allows an application to read SMS messages.". For this type of API, we try to train a text similarity detection model to find these hidden APIs by calculating the similarity between their descriptions and dangerous permissions' descriptions.

We use LSTM Siamese Network [23] like Fig 3 to calculate the similarity of description. Training the siamese network requires pairs of description. To prepare positive and negative pairs, firstly, we group the dangerous_permissions and APIs which require the permissions into different groups, which was shown in TableII. After that, we consider the permissions and APIs in the same group as similar(positive) pairs; permissions and APIs in the different groups are not similar (negative) pairs. By training these positive and negative examples using the Siamese network, we can obtain a prediction model for comparing the similarity of the permission and API functions.

IV. EXPERIMENT

In this section, we design a set of experiments to evaluate the performance of the API features we select and make a comparison with the state-of-the-art in terms of the four research questions listed in the Introduction. Moreover, we make a detailed discussion of all approaches used to solve the multi-family classification problem. All experiments are conducted on an Intel (R) Core (TM) i7-4790 CPU PC running Windows 7 SP1 (64 bit) with Keras [7].

A. Datasets

We carry out our experiments on three well-known datasets of Android malware families: Android Malware Genome Project (Genome), Drebin Dataset (Drebin), and AMD Project (AMD). Genome collects 1260 malware applications that cover 49 malware families. Drebin contains 5560 Android applications from 179 different malware families. AMD contains 24553 malware applications, which is categorized in 72 malware families.

For the apk file that cannot be decompiled, we delete it from the samples. We also removed the category and its samples with only one sample in the three datasets because one sample cannot be divided into the training set, the verification set, and the test set. After that, we get three preliminary screened datasets for our experiments. M1(Genome) contains 1246 malware applications in 30 Android malware families. M2(Drebin) contains 5426 malware applications in 132 malware families, and M3(AMD) contains 24507 malware applications in 71 malware families.

For all classification experiments, we use the 10-fold cross-validation method to divide the three data sets. For families with a sample size of less than 10, the leave-one-out method is used to ensure that N-fold cross-validation data can be obtained. Then, we randomly copy the 10'N fold of these N folds. The original N-fold and the repeated 10'N-fold constitute the 10-fold cross-validation data of a small family. In this way, we guarantee that there is at least one sample from each family in the test data. So far we have completed the division of the data set.

B. Candidate API List and Input

In this section, we will describe the construction process of the model input. As we adopt syntactic features(Android permissions, critical APIs) as the features of Malware. We use apktool to decompile Android products through reverse engineering and then obtain these syntactic features from XML files and smali files. Then we adopt a one-hot vector to represent the features where each vector element represents whether there are syntactic features in the application. In this way, we will convert an apk file to an input vector of a machine learning model.

C. Evaluation Metrics

Given that the multi-class classification needs to present the classification results of each class, we adopt F1-score which is widely used to evaluate the classification results in [26], [35], [34], [31]. **F-score** for each family is the harmonic mean of recall and precision. Here we choose 1 as the value of β in F-score computation, so F1-score is the primary evaluation metric in the experiment. The overall metric is the mean of the four metrics.

D. Hyperparameters

Iteration. Generally, more iterations mean more train times, which will bring better accuracy during the neural network training. However, more iterations also cost more time. To balance the accuracy and time cost, we set the number of iterations from 1 to 2000. Meanwhile, the *error rate*, which equals 1 - *accuracy rate*, is used to show the performance of classification. According to the experiment results, we finally set the number of training iteration to 50.

Batch Size. In neural networks, the batch size defines the number of samples fed into the network for training each

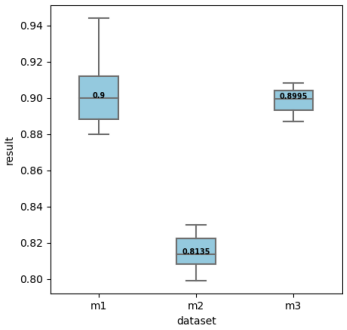

Fig. 4. Result of MLP classifier via first hop APIs.

time. Batch size has an impact on training time and accuracy. Choosing the right batch size will save time and complete the training task with high quality. We set 256 batch sizes for training after experimenting with the batch size with 32, 64, 128, 256, and 512.

Learning Rate. The learning rate sets the weight update rate of the neural network. Considering that the number of neural network layers used in our paper is not large, the structure is not complicated, so a fixed learning rate is used here. After experimenting this parameter with the value {0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001} we choose 0.0005 as learning rates.enting with the batch size with 32, 64, 128, 256, and 512.

E. Research Questions and Findings

We design a set of experiments to address the following research questions (RQs) in this subsection.

RQ1: Is it appropriate to use APIs that are directly related to the permission whose protection-level is dangerous?

Through API documentation, we obtained a total of 30 dangerous permissions. After that, we searched the APIs directly related to the 30 dangerous permissions and obtained 121 candidates' sensitive APIs through the knowledge graph. After that, we will use these 151 features as syntactic features and use an MLP classifier to test the effectiveness of these syntactic features on Android malware classification on three data sets.

TABLE III
CLASSIFICATION ACCURACY OF STATE-OF-THE-ART APPROACHES.

Approach	ACC
MudFlow [3]	0.881
DroidLegacy [10]	0.929
FalDroid [11]	0.972
Apposcopy [12]	0.900
Dendroid [27]	0.942
Astroid [13]	0.938
DroidSIFT [36]	0.930

Fig. 4 shows the result of the MLP classifier, which uses the first searched APIs on three Malware datasets. It seems that the model has good classification results in the three data sets. Moreover, it is similar to the state-of-art method in TableIII. Compared with the m1 and m3, the prediction result of m2

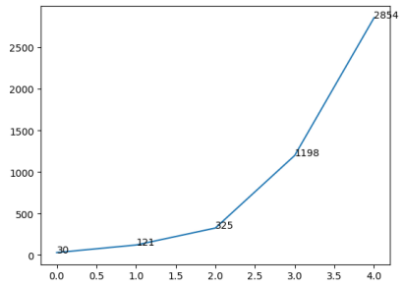

Fig. 5. The number of candidate APIs obtained by multiple hops.

is relatively lower, which may be related to the m2 data set itself. However, the overall prediction result is satisfactory. Compared with other methods using the m1 data set, the result of using MLP as a classification is very close to the existing benchmark. It may be because the number of APIs for the first link is small and not comprehensive, so the predicted results are not perfect. The result is not particularly ideal because just using directly related APIs is not sufficient to meet the requirements of classification tasks. It can be improved by adding indirectly related APIs. Nevertheless, the results close to the existing methods illustrate this method's effectiveness, which uses the API Knowledge Graph to search APIs linked to dangerous permission as a syntactic feature is sufficient for classification.

> *We can get sensitive API lists from API Knowledge Graph and it is somewhat appropriate to use APIs that are directly related to permissions with dangerous levels of protection.*

RQ2: How many APIs are indirectly related to permission whose protection-level is dangerous is sufficient for classification tasks?

In addition to the 121 sensitive API accidents directly related to dangerous permission, we can obtain more APIs that are indirectly related to dangerous permission through knowledge graph relationship search. Moreover, as the relationship passes, the number of acquired APIs will increase rapidly. Whether getting more APIs through more links improve the classification results, and how many orders of magnitude are sufficient for malware classification tasks?

Fig. 5 shows the growth of the number of candidate APIs obtained by multiple jumps. We can see that the 30 dangerous permissions from the beginning, after four relationship passes, got 2854 candidate APIs. The number increased by 90 times. Therefore, it is not possible to increase the number of links without restrictions to increase the number of related APIs.

We conducted a malware classification test on the list of candidate sensitive APIs obtained from different relationship transfers. We observed whether the classification effect would improve with the number of APIs introduced.

Fig. 6 shows the results on M1, M2, M3 using different lists of candidate sensitive APIs obtained by different transfer

Fig. 5 shows the growth of the number of candidate APIs obtained by multiple jumps. We can see that the 30 dangerous permissions from the beginning, after four relationship passes, got 2854 candidate APIs. The number increased by 90 times. Therefore, it is not possible to increase the number of links without restrictions to increase the number of related APIs.

We conducted a malware classification test on the list of candidate sensitive APIs obtained from different relationship transfers. We observed whether the classification effect would improve with the number of APIs introduced.

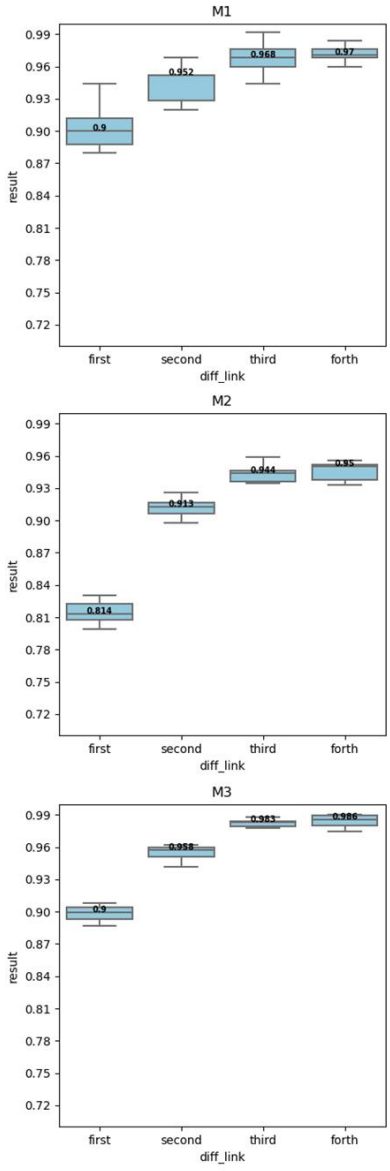

Fig. 6. Results for different hops on M1, M2 and M3.

times of relationship as a syntactic feature.

We can see from the figure that as the number of links increases, the prediction results' accuracy also increases. Compared with the first-link results, the results of the second-link are significantly improved on the three data sets, which has a 5% increase on m1, a 10% increase on m2, 5% increase on m3. Moreover, the third-link results also raise based on the second-link results, but it is not as apparent as the improvement from first-link to second-link, which has a 1.6% increase on m1, 3.1% increase on m2, and 2.5% increase on m3.

However, for forth-link results, they have not been improved much compared to third-link results, only 0.2% on m1, 0.6% on m2, and 0.3% on m3.

It can be seen from the results that using the APIs obtained by Hyperlink multiple times as malware features can further improve the classification performance, and the fourth link has reached its peak. Besides, the classification results of m2 in the forth-link are no longer significantly different from m1 and m3.

> *We can get more sensitive APIs through the API knowledge graph, but when the number of the relationship passes to three times, the resulting list of sensitive APIs can already be competent for malware classification tasks.*

RQ3: Is it possible to discover hidden sensitive APIs that cannot be obtained by searching the API knowledge graph?

We trained an LSTM-siamese model to calculate the similarity between the description of the permission and the API's functional description.

We have collected nearly 80,000 APIs. Searching in such a large area is tedious and time-consuming (not all classes are related to sensitive operations). Therefore, we focus on the APIs declared by the classes obtained from the APIs searched in **RQ2**.

In this way, we obtained 2757 different APIs that have never appeared in the four links. To test the effectiveness of these hidden APIs in classifying malware, we added them to the APIs used in **RQ2**. Compare the results of the classification to check whether the remote APIs found are helpful in malware classification. Fig. 7 show the classification results of different link APIs after adding hidden APIs on three datasets. Blue boxes show the result based on the hyperlink, and green boxes show additional hidden APIs.

The classification results with additional hidden APIs showed an overall growth trend and were similar to the growing trend of APIs using the only hyperlink. There is a significant improvement from first-link to second-link (4% on m1, 5.7% on m2, 2.7% on m3) a small improvement from second-link to third-link(0.4% on m1, 1.6% on m2, 0.9% on m3), and no significant change from third-link to forth-link.

Compared with the classification results using only the APIs obtained by hyperlink as features, the results with additional hidden APIs have visible improvement in each link's results.

This result shows that additional hidden APIs improve the classification results and prove that the API obtained by calculating the similarity of the functional description as a feature also helps the malware classification to a certain extent.

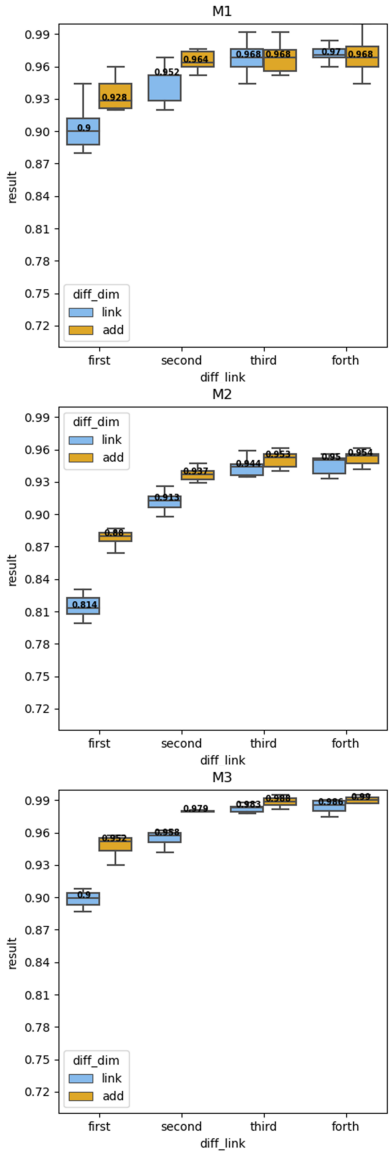

Fig. 7. Results of comparison between hyperlink feature with similarity feature for different hops on M1, M2 and M3.

We can compare the similarity of the functional description part of the API to discover the APIs in the graph that cannot be found through a hyperlink, and these APIs have improved the malware classification results to a certain extent.

RQ4: Which permission_group related APIs are more helpful for malware classification?

After examining the effectiveness of these APIs in classifying malware, we tried to find which permission_group of APIs were more helpful for classifying malware.

The difference from the previous experiment is that we divide the obtained APIs into groups according to TableII. Use the APIs of each group as malware features to conduct malware classification experiments. Because we did not find APIs related to their members from these three groups**ACTIVITY_RECOGNITION, CALENDAR, SENSORS** from API Knowledge Graph, we did not use them in the classification task. After deleting these three groups, we performed a classification comparison of the remaining eight groups.

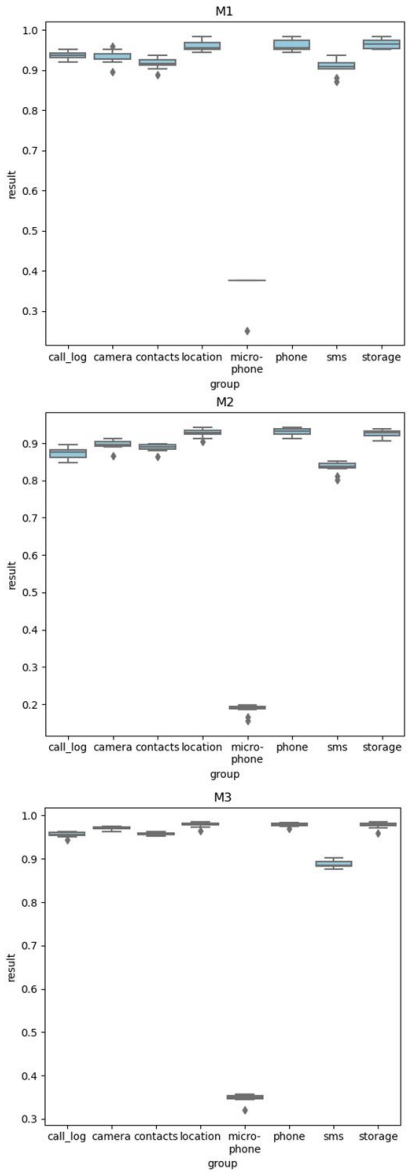

Fig. 8. Results of different permission groups.

Fig. 8 shows the classification results of the different groups on three datasets.

First, in the three data sets, each group's classification results did not exceed the results of the data set using the four hyperlink API as a feature. This shows that using APIs related to a single dangerous_permission group as features is not complete.

Besides, we can see that the box plot contains many outliers.

These outliers indicate that using a single group of APIs as features for classification will result in inconsistent classification results, and the classification results will fluctuate greatly under 10-fold cross-validation. This result also illustrates the importance of using multiple groups of dangerous_permission group.

In Fig. 8, we also observe that three groups of API features perform well on all three datasets. They are **LOCATION, PHONE, STORAGE**. The APIs related to them are superior in number, and the operations designed by these three permission_groups are more related to user privacy operations(device location, telephony features, and storage). Moreover, the microphone group has the worst-performing. There are too few APIs related to it. Furthermore, microphone audio may not involve risky operations.

*The permissions in group **LOCATION, PHONE, STORAGE** perform better than others. APIs related to them are more suitable for classification tasks and maybe closer to user privacy operations. The permission in group **MICROPHNE** performs worst. APIs related to them may not be too helpful for classification tasks.*

V. Threats To Validity

In this section, we discuss several threats that may affect the validity of our experiment results.

A. KGRelation

Although we have achieved well family classification performance on the three Android malware data sets, the API relationships we use are all based on Hyperlink. However, in the description given in the API official document, the description section also includes API and other The more detailed relationship of API, such as calling, being called, and common use. We will add this part of the work in the follow-up work to clarify the relationship between APIs and remove redundant relationships to ensure the quality of the API knowledge graph.

B. Similarity Calculation

We consider calculating the functional description similarity of two APIs to expand the strange relationship in the API knowledge graph in the similarity calculation task. However, we consider that it can also be used to assist the task of reducing redundant relationships. If the similarity of the two APIs is lower than the threshold, we can consider removing the hyperlink relationship.

VI. CONCLUSION

In this paper, we conduct extensive experiments for selecting APIs, which is better for Android malware family prediction as features. To filter out suitable APIs as features, we built an API knowledge graph and selected APIs related to dangerous permissions for malware classifications based on the dangerous permission groups required in Android development and the hyperlink relationship in the knowledge graph. After that, to test the selected API's effectiveness in the classification task, we conducted many experiments based on MLP. We used the classification results to reflect the role of API as a feature. Comparing the F1 value of the classification result, using the API we selected as a feature for classification has achieved an excellent classification effect. Not only that, after adding remote APIs that were found by comparing the similarity of API function describe, the classification results have also been significantly improved. This illustrates the effectiveness of finding an API suitable for malware classification through dangerous permission. Meanwhile, we also find that APIs related to the three dangerous_permisison groups PHONE, Location, and STORAGE can help malware classification.

Acknowledgement

We appreciate the anonymous reviewers for their beneficial feedback. This work has partially been sponsored by the National Science Foundation of China (No. 61872262, 61572349).

References

[1] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[3] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 426–436. IEEE, 2015.

[4] Alberto Bacchelli, Marco D'Ambros, Michele Lanza, and Romain Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *2009 16th Working Conference on Reverse Engineering*, pages 205–214. IEEE, 2009.

[5] Alberto Bacchelli, Michele Lanza, and Vitezslav Humpa. Towards integrating e-mail communication in the ide. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, pages 1–4, 2010.

[6] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[7] Francois Chollet. Keras. Website, 2015. https://github.com/keras-team/keras.

[8] Ronan Collobert and Samy Bengio. Links between perceptrons, mlps and svms. In *Proceedings of the twenty-first international conference on Machine learning*, page 23, 2004.

[9] Barthélémy Dagenais and Martin P Robillard. Recovering traceability links between an api and its learning resources. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 47–57. IEEE, 2012.

[10] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014*, pages 1–12, 2014.

[11] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 13(8):1890–1905, 2018.

[12] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587, 2014.

[13] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. *arXiv preprint arXiv:1608.06254*, 2016.

[14] IDC. Smartphone market share. Website, 2019. https://www.idc.com/promo/smartphone-market-share/os.

[15] JAVA. Permission. Website. https://developer.android.google.cn/reference/android/R.attr#protectionLevel.

[16] JAVA. permissiongroup. Website, 2020. https://developer.android.google.cn/reference/android/Manifest.permission_group.

[17] He Jiang, Liming Nie, Zeyi Sun, Zhilei Ren, Weiqiang Kong, Tao Zhang, and Xiapu Luo. Rosf: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing*, 12(1):34–46, 2016.

[18] Deokyoon Ko, Kyeongwook Ma, Sooyong Park, Suntae Kim, Dongsun Kim, and Yves Le Traon. Api document quality for resolving deprecated apis. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 2, pages 27–30. IEEE, 2014.

[19] Miroslav Kubat. Neural networks: a comprehensive foundation by simon haykin, macmillan, 1994, isbn 0-02-352781-7. *The Knowledge Engineering Review*, 13(4):409–412, 1999.

[20] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193. IEEE, 2018.

[21] McAfee. Mcafee labs threats report. Website, 2018. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2018.pdf.

[22] Paul W McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290, 2014.

[23] Jonas Mueller and Aditya Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *thirtieth AAAI conference on artificial intelligence*, 2016.

[24] Martin P Robillard and Yam B Chhetri. Recommending reference api documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[25] Christopher Scaffidi. Why are apis difficult to learn and use? *XRDS: Crossroads, The ACM Magazine for Students*, 12(4):4–4, 2006.

[26] Stephen V Stehman. Selecting and interpreting measures of thematic classification accuracy. *Remote sensing of Environment*, 62(1):77–89, 1997.

[27] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[28] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652, 2014.

[29] Jiamou Sun, Zhenchang Xing, Xin Peng, Xiwei Xu, and Liming Zhu. Task-oriented api usage examples prompting powered by programming task knowledge graph. *arXiv preprint arXiv:2006.07058*, 2020.

[30] Gias Uddin and Martin P Robillard. How api documentation fails. *IEEE Software*, 32(4):68–75, 2015.

[31] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[32] Wei Wang, Xing Wang, Dawei Feng, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. Exploring permission-induced risk in android applications for malicious application detection. *IEEE Transactions on Information Forensics and Security*, 9(11):1869–1882, 2014.

[33] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[34] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering*, 42(10):977–998, 2016.

[35] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 51–62. IEEE, 2016.

[36] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1105–1116, 2014.

[37] Yanxin Zhang, Yulei Sui, Shirui Pan, Zheng Zheng, Baodi Ning, Ivor Tsang, and Wanlei Zhou. Familial clustering for weakly-labeled android malware using hybrid representation learning. *IEEE Transactions on Information Forensics and Security*, 15:3401–3414, 2019.

[38] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622, 2013.

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

[40] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.