Computers & Security 137 (2024) 103651

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

A novel Android malware detection method with API semantics extraction

Hongyu Yang [a,*], Youwei Wang [a], Liang Zhang [b,*], Xiang Cheng [c], Ze Hu [a,*]

[a] Civil Aviation University of China, Jinbei Road No. 2898, Dongli, 300300, Tianjin, China
[b] The University of Arizona, East University Boulevard No. 1200, Tucson, 85721, AZ, USA
[c] Yangzhou University, Daxue South Road No. 88, Yangzhou, 225127, Jiangsu, China

ARTICLE INFO

Keywords:
Evolved malware detection
Model aging
API semantics
Function call graph
Machine learning

ABSTRACT

Due to the continuous evolution of both the Android framework and malware, conventional malware detection methods that have been trained using outdated apps are inadequate in effectively identifying sophisticated evolved malware. To address this issue, in this paper, we propose a novel Android malware detection method with API semantics extraction (AMDASE), it can effectively identify evolved malware instances. Firstly, AMDASE performs API clustering to obtain cluster centers representing API functions before malware detection. We design API sentence to summarize API features and employ natural language processing (NLP) tools to acquire embeddings of API sentence for clustering. With the help of API sentence, it becomes possible to effectively extract the semantics of API contained in features like method name that accurately represents its intended functionality, which also makes the clustering results more accurate. Secondly, AMDASE extracts call graph from each app and optimizes the call graph by removing nodes corresponding to unknown functions, while ensuring the preservation of connectivity between their predecessor and successor nodes. The optimized call graph can extract more robust API contextual information that accurately represents the behavior of each app. Thirdly, in order to maintain resilience against the evolution of Android malware, AMDASE extracts function call pairs from the optimized call graph and abstracts the APIs in function call pairs into cluster centers obtained in API clustering. Finally, feature vectors are generated using one-hot mapping and machine learning classifiers are used for malware detection. We evaluate AMDASE on a dataset of 42,154 benign and 42,450 malicious apps developed over a seven-year period. The experimental results demonstrate that AMDASE greatly outperforms the existing state-of-the-art methods and has a significantly slower aging speed.

1. Introduction

The vast popularity of the Android platform has fueled the rapid expansion of Android malware. Although machine learning technology has achieved excellent results in malware detection (Qiu et al., 2020), the continuous evolution of malware still poses great challenges to the detection system (Jordaney et al., 2017). It is difficult for a classifier trained with outdated apps to effectively detect new apps that appear after a certain period of time, resulting in a continuous decline in the detection performance over time, also known as model aging.

Model aging is caused by concept drift, unlike computer vision or natural language processing (NLP), the problem domain of malware detection changes frequently, which leads to the difference in the feature distribution between newly emerged malware and older malware. Malware developers often evade detection systems by altering implementations to maintain malicious behavior (Zhang et al., 2020). Specifically, the evolved malware has a large number of APIs that did not appear in the training set.

Currently, most malware detection methods are based on information of API invocations. The detection method based on API frequency information (Yang et al., 2014; Arp et al., 2014; Feng et al., 2020) exhibits a high false negative rate and is vulnerable to malicious attacks. The detection method that relies on the API contextual information (Wu et al., 2019; Allen et al., 2018) demonstrates enhanced accuracy in identifying malicious behavior exhibited by apps. However, it faces a challenge in effectively extracting the API contextual information due to the absence of a suitable approach for handling a substantial volume of unknown function nodes within the call graph. In addition, these existing detection methods fail to consider the frequent changes of APIs, thereby posing challenges in effectively identifying evolved malware.

API clustering groups APIs with similar functions into the same category. During the malware detection, the substitution of APIs in the

* Corresponding authors.
E-mail addresses: yhyxlx@hotmail.com (H. Yang), liangzh@arizona.edu (L. Zhang), zhu@cauc.edu.cn (Z. Hu).

https://doi.org/10.1016/j.cose.2023.103651
Received 15 August 2023; Received in revised form 28 October 2023; Accepted 8 December 2023
Available online 12 December 2023

feature vector with API cluster centers can be employed to enhance the classifier's ability to recognize APIs that were not encountered during the training phase. This approach has the potential to mitigate the rate at which the classifier's performance deteriorates over time (Zhang et al., 2020). However, existing API clustering methods extract API semantics from features such as package, parameters, and return value, while neglecting crucial features like method name and permission.

The limited enhancement of classifier's performance through API clustering can be attributed to the inadequacy of extracting API semantics. Mariconti et al. (2016) proposed MAMADROID, wherein they employed a clustering technique to group APIs based on their package names. MAMADROID demonstrates resilience to newly introduced APIs within existing packages by abstracting them into packages. Nevertheless, it is important to note that APIs contained within the same package may not necessarily exhibit similar functionalities. Moreover, MAMADROID cannot capture any information from APIs in the new package. Pendlebury et al. (2019) found that under the experimental setting of eliminating time deviation, the *F1*-Measure of MAMADROID dropped from over 90% to 58% after only three months.

Aiming at addressing the aforementioned deficiencies in existing malware detection methods, we propose a novel Android malware detection method with API semantics extraction (AMDASE) that can effectively detect evolved malware. Firstly, we perform API clustering to obtain cluster centers representing API functions before malware detection. We design API sentence to summarize API features and employ NLP tools to acquire embeddings of API sentence for clustering. Secondly, we extract call graph from each app and optimize the call graph by removing unknown function nodes while preserving the connection between their predecessor and successor nodes. Thirdly, we extract function call pairs from the optimized call graph and abstract APIs in the call pairs into the cluster centers obtained in the API clustering part. Finally, feature vectors are generated using one-hot encoding and machine learning classifiers are used for malware detection.

We evaluate the detection performance of AMDASE on a dataset of 42,154 benign apps and 42,450 malicious apps developed over a seven-year period. AMDASE trained on data from 2012 to 2013, achieves an average *F1*- Measure of 82.6% when detecting samples from 2014 to 2018. The average *F1*- Measure represents a 22% increase in comparison to the state-of-the-art malware detection method, MAMADROID. Our experimental results demonstrate that AMDASE greatly outperforms the existing detection methods and has a significantly slower aging speed.

This paper makes three primary contributions:

1. Firstly, we propose a semantic distance based API clustering approach. We design API sentence to summarize the characteristics of API. API sentence not only contains important features like method name that outlines the functionality of the API but also uniformly maps APIs with an inconsistent number of features into feature vectors with fixed-size. With the help of API sentence, it becomes possible to effectively extract the semantics of API, which makes the API clustering results more accurate.
2. Secondly, we introduce a call graph optimization method. This approach removes all unknown nodes while maintains the connection between their predecessor and successor nodes. If there exists a path consisting of all unknown function nodes between any two API nodes (e.g., APIx and APIy) in the call graph, APIx directly calls APIy in the optimized call graph. Call graph optimization enables the extracted API contextual information to more precisely reflect the app's behavioral patterns.
3. Finally, we present AMDASE, a novel Android malware detection method that can effectively identify evolved malware without requiring any kind of retraining. By abstracting APIs into the cluster centers, AMDASE is resilient to the API changes in both Android framework and malware.

2. Related work

Machine learning has achieved significant advancements in Android malware detection (Wu et al., 2023; Bhat et al., 2023; Tarwireyi et al., 2023; Yang et al., 2015; Zhang et al., 2014; Yang et al., 2014; Karbab et al., 2018; Chen et al., 2018; Feng et al., 2020; Wang et al., 2019, 2018; Liu et al., 2019; Lu et al., 2021b). Nevertheless, the issue of model aging has consistently posed a significant challenge. Pendlebury et al. (2019) and Jordaney et al. (2017) detail that the phenomenon of concept drift is prevalent in Android malware detection. Researchers have put forth numerous solutions to address this issue, which mainly can be divided into four categories: retraining, app relation graph, abnormal sample identification, and API clustering.

2.1. Malware detection based on retraining

The term 'retraining' pertains to the process of training the classifier anew by utilizing concept drift samples along with their corresponding labels. According to different sources of the label, the process of retraining can be categorized into two types: real label retraining and pseudo label retraining. The real label comes from manual labeling by malware experts. The term 'pseudo label' refers to the inferred class labels generated by malware classifiers, typically characterized by a significant level of confidence in their predictions.

Grosse et al. (2017) proposed an adaptive and scalable Android malware detection method DroidOL. The DroidOL system utilizes the call graph to extract sensitive behavioral features of an app. It employs misclassified samples, along with their corresponding real labels, to update the classifier during the detection process. Hence, DroidOL possesses the capability to adjust to the progression of Android apps.

Xu et al. (2019) proposed an online learning based malware detection method DroidEvolver. DroidEvolver believes that classifiers trained with different optimizers have different aging speeds, so a model pool consisting of five linear online learning algorithms is constructed during the training phase. In the detection phase, the aged model is updated using the prediction results of other un-aged models.

Detection methods based on retraining generally have the following deficiencies: The process of retraining a real label necessitates a substantial quantity of concept drift samples that are of high quality, a task that proves difficult to accomplish within a limited time frame. In addition, the process of retraining actual labels necessitates a substantial allocation of human resources for the manual labeling of concept drift samples. Pseudo label retraining obviates the need for manual sample labeling by human resources, yet it is accompanied by two inherent limitations. First is that once the predicted label is incorrect, it will bring a steep decline in the performance of the classifier. Another drawback is that malicious attackers can design specific samples to utilize the mechanism of pseudo label retraining.

2.2. Malware detection based on app relation graph

App relation graph refers to constructing an entity-relation graph with each app as its main node to reflect the similarity between different apps. The Android ecosystem undergoes a gradual evolutionary process rather than sudden mutations, resulting in evolved malware that retains significant similarities to their ancestral counterparts. Once a concept drift app is misclassified, classifiers may use the similarity between this app and its ancestor nodes in the app relation graph to pull it back to the decision boundary.

Gu and Li (2021) proposed a method to alleviate model aging in Android malware detection. It builds an app relation graph to reflect the evolutionary relationship between apps and this graph assists classifier for malware detection.

Hei et al. (2021) presented a heterogeneous graph attention network based Android malware detection method. In the training phase, an app relation graph was constructed to simulate the similarity between

apps. Then, a graph neural network was used to learn the similarity between apps and generate feature vectors for malware detection. In the detection phase, an incremental aggregation method MsGAT++ was designed, which enables the rapid generation of feature vectors without the need to update the entire app relation graph.

Detection methods based on app relation graph generally have the following deficiencies: First, code reuse is an important part of software development, as it allows developers to utilize existing code for various purposes. However, it is worth noting that the practice of code reuse is not limited to benign apps, as malicious actors may also employ reused code from benign apps in the development of malware. This brings serious difficulties to malware detection methods utilizing the similarities between apps. At the same time, the similarity between apps does not reflect their behavior, which poses a challenge for detection methods based on app relation graphs to learn the differences in behavior features between malware and benign apps.

2.3. Malware detection based on abnormal sample identification

Abnormal sample identification is filtering out the concept drift samples during the detection. The decision made by malware classifiers for these samples usually has low confidence, thereby necessitating the involvement of malware experts to assess and make informed judgments regarding these apps.

Yuan et al. (2022) proposed a method for concept drift sample identification using a double-headed neural network. This neural network has two parallel output layers which output the prediction result of app respectively. The concept drift samples are identified by the difference between these two output layers.

Karbab and Debbabi (2021) proposed an adaptive Android malware detection method. Classifiers usually have low confidence when making decisions on concept drift samples, so low-confidence samples are filtered during malware detection.

Abnormal sample identification can greatly improve the effectiveness of the detection method, but it is limited to identifying concept drift samples, rather than evolved malware detection.

2.4. Malware detection based on API clustering

API clustering groups APIs with similar functions into same category. During the malware detection, the substitution of APIs in the feature vector with API cluster centers enhances the resilience of detection method against the frequent alterations of APIs.

MAMADROID (Mariconti et al., 2016) clusters API by package name. By abstracting the API calls into packages, MAMADROID is resilient to newly introduced APIs that appeared in existing packages. However, APIs within the same package do not necessarily perform similar functions (Zhang et al., 2020). Moreover, MAMADROID cannot capture any information from APIs in new packages.

Zhang et al. (2020) proposed APIGRAPH, an enhancement to existing Android malware classifiers via API clustering. Firstly, APIGRAPH extracts information from Android official documentations to build an entity-relation graph. Then the embedding vectors of entities and relations are generated using the TransE (Van der Maaten and Hinton, 2008) algorithm. Finally, API clustering is performed using the k-Means (Syakur et al., 2018) algorithm. The authors believe that APIs with similar parameters, similar permissions, similar packages, and similar return values must have similar functions. However, in the process of API semantics extraction, method name of API which contains numerous semantic information is ignored.

Lei et al. (2019) introduced an API clustering method using the tokens of API. Initially, it splits each API within the call graph into tokens, encompassing its family, package name, class name, method name, return value, and parameters. Then, doc2vec (Lau and Baldwin, 2016) is used to encode and generate the embeddings of each API. Finally, clustering is completed using the k-Means algorithm. This API clustering method uses method name during API semantics extraction, but critical features like permission and exception are ignored. Permission is extremely important in Android system (Au et al., 2012). At the same time, this method collects APIs from training set rather than API documentations, which makes this method unable to identify new APIs appeared in testing set.

Xu et al. (2020) proposed an API clustering method using the API contextual information in the call graph. This method posits that APIs exhibiting comparable patterns of callers and callees are likely to possess analogous functionalities. Therefore, a two-layer neural network was designed to capture the potential relationship between API invocations and its contextual information. The hidden layer in the neural network is used as embeddings reflecting API semantics for clustering. However, this method uses training data and testing data together to train the neural network, which is a serious data snooping (Quiring et al., 2022) behavior. In actual scenarios, this method cannot identify new APIs appeared in testing set.

Compared with other methods, API clustering is the most direct and effective solution to evolved malware detection. However, existing clustering methods ignore vital features like method name and permission during API semantics extraction, resulting in limited performance improvement of classifiers by clustering results.

3. Overview of AMDASE

AMDASE is a static malware detection method using API contextual information. The system exhibits a robust capacity for detecting evolved malware. Fig. 1 illustrates the framework of AMDASE. It consists of two components: the semantic distance based API clustering and the Android malware detection.

Semantic distance based API clustering generates API cluster centers representing the function of each API, which is finished before Android malware detection. There are four steps in the semantic distance based API clustering. Firstly, the features of each API are extracted from the Android official documentations (Api documentations, 2023) (Step a in Fig. 1), including exceptions, permissions, parameters, returns, etc. Secondly, API sentences are generated based on the prescribed regulations (Step b in Fig. 1). Thirdly, the API sentence is encoded using Bert (Jacob et al., 2019), a well-established NLP model, to acquire the embeddings of each API (Step c in Fig. 1). Finally, we use k-Means algorithm to generate cluster centers representing API function (Step d in Fig. 1) and group APIs with similar semantics together.

Android malware detection process consists of five steps. To begin, FlowDroid (Arzt et al., 2014), a static analysis tool, is employed to extract the function call graph from each app (Step 1 in Fig. 1). Additionally, the call graph is optimized by eliminating nodes representing unknown functions, while ensuring that the connections between their predecessor and successor nodes are maintained (Step 2 in Fig. 1). Furthermore, the function call pairs within the call graph are extracted (Step 3 in Fig. 1). In the fourth step, the function call pairs are initially abstracted into pairs of API clusters. Next, we proceed to abstract the remaining methods into their respective packages. Finally, one-hot encoding is employed to produce a feature vector for each app (Step 4 in Fig. 1). In the last step, the feature vector is fed into machine learning classifiers to predict the app as either malicious or benign (Step 5 in Fig. 1).

4. Semantic distance based API clustering

There are four parts of semantic distance based API clustering: API feature extraction, API sentence generation, API sentence encoding, and cluster center generation.

4.1. API feature extraction

Feature extraction is to extract the features of each API from Android official documentations (Api documentations, 2023). The features em-

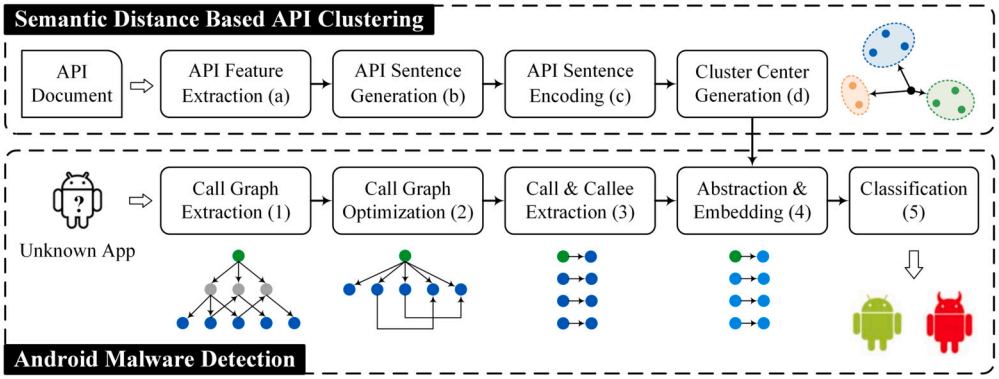

Fig. 1. Architecture of AMDASE.

Table 1
Example of extracted API features.

Android.telephony.TelephonyManager.getDeviceID	
Parameter:	None
Return:	None
Exception:	None
Permission:	READ_PHONE_STATE
Package:	Android $ telephony
Class name:	Telephony $ Manager
Method name:	get $ Device $ ID

Table 2
Rules of unique part sentence generation.

Feature	At Least One Feature Value (R_1)	None Feature Value (R_0)
Permission	use permission *permission_1*, ...	use none permission
Exception	throw exception *exception_1*, ...	throw none exception
Parameter	use parameter *parameter_1*, ...	use none parameter
Return	return *return_value*	return none

ployed in our study encompass seven distinct categories: package, class name, method name, parameter, permission, exception, and return. The authors of APIGRAPH provided the collected Android official documentations and the source code of API feature extraction (Apigraph, 2023). By reproducing the source code, we were able to obtain all APIs and their corresponding features.

Existing API clustering methods ignore method name when API semantics extraction, however, the method name of API contains numerous information and sketches the function of API. By employing NLP models, it is possible to effectively extract the semantic information embedded within the method name of an API, thereby providing an approximation of its intended functionality. For example, APIs with method name such as 'getDeviceId' or 'setWifiEnabled'. The names of their methods allow us to easily comprehend their functions.

Table 1 shows the feature extraction outcome for API called *Android.telephony.TelephonyManager.getDeviceId*. Note that to reduce the impact of some widely used feature values on API semantics extraction, we ignore feature values such as *int*, *boolean*, *String*, and *float*. The API shown in Table 1 takes *int* as a parameter and returns *String*. These two feature values are ignored during the feature extraction.

In addition to the existing feature extraction methods in APIGRAPH, we modify API features by separating the method name and class name into words utilizing their camel case (except the first word, the first letters of the rest words are all uppercase, such as 'setWifiEnabled' and 'sendTextMessage'). The method name of API shown in Table 1 is 'getDeviceID', which can be divided into three words: 'get', 'Device', and 'ID', split by '$'. Note that some words in class names and method names are all uppercase, such as 'SQL', 'URL', and the 'ID' in Table 1, such words are abbreviations and require additional processing.

4.2. API sentence generation

After the feature extraction is completed, we convert APIs into embeddings that can reflect their semantic information. The problem in this part is that the value number of the same feature is inconsistent. For example, the method name of some APIs can be split into five words, while some APIs can only be split into three words. Some APIs require permissions for invocation, while others do not. Therefore, it is necessary to design a method of embedding that can not only represent the semantic information of API but also uniformly maps all APIs with a variable number of feature values into feature vectors with fixed-size, which are finally used for distance calculation in subsequent clustering.

To solve this problem, we first convert each API and its features into a sentence, called API sentence. According to the number of feature values, the generation of API sentence is divided into common part and unique part. The common part refers to three common features that all APIs have and only have one value: method name, class name, and package. The rule of common part sentence generation is below:

$$\text{method } Method \text{ from class } Class \text{ from package } Pack$$

There are four types of features in the unique part: permission, exception, parameter, and return. When it comes to these unique features, each API has a different number of feature values. Table 2 lists the rules of unique part sentence generation.

The pseudo-code of API sentence generation is shown in Algorithm 1. The input is seven types of features of APIx, and the output is the API sentence S of APIx. Line 1 to 3 generate the corresponding part of common features in the API sentence; Lines 4 to 7 call Algorithm 2 to generate the corresponding part of unique features in the API sentence; line 8 replaces all uppercase letters in S with lowercase letters; line 7 replaces each '$', '_', and '.' in S with a space.

Algorithm 1 API Sentence Generation.

Input: Seven types of features of APIx, including $Method$, $Class$, $Pack$, $Permission$, $Exception$, $Parameter$, and $ReturnValue$
Output: API sentence S of APIx
1: S = "method " + $Method$
2: $S = S$ + " from class " + $Class$
3: $S = S$ + " from package " + $Pack$
4: $S = S + Algorithm2$ ($Permission$, " use none permission", " use permission")
5: $S = S + Algorithm2$ ($Exception$, " throw none exception", " throw exception")
6: $S = S + Algorithm2$ ($Parameter$, " use none parameter", " use parameter")
7: $S = S + Algorithm2$ ($ReturnValue$, " return none", " return")
8: replace all uppercase in S by lowercase
9: replace all '$', '_', and '.' in S By ' '
10: **return** S

Algorithm 2 shows the pseudo-code of unique feature processing in API sentence generation. Algorithm 2 is called by lines 4 to 7 in Algorithm 1. The input is a feature value list $Features$ of a unique feature

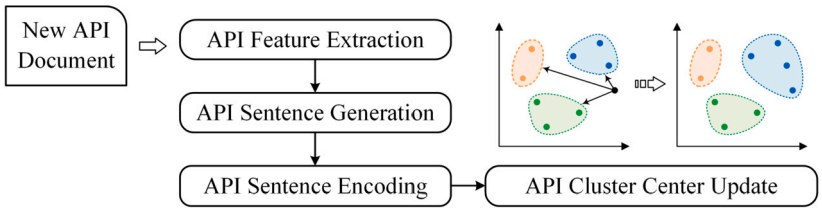

Fig. 2. Updates of API clustering results.

and the corresponding API sentence generation rules R_0 and R_1. The output is a part of the API sentence S_{Part} corresponding to the unique feature. Algorithm 2 considers two cases where the number of feature values is 0 or at least 1.

Algorithm 2 Unique Feature Processing.

Input: Feature value list $Features$ of a unique feature; Corresponding API sentence generation rules R_0 and R_1 of the unique feature
Output: A part of API sentence S_{Part} corresponding to the unique feature

```
if Features is none then
    S_Part = R_0
else
    S_Part = R_1
    for each each feature f in Features do
        S_Part = S_Part + ' ' + f
    end for
end if
return S_Part
```

After inputting the API shown in Table 1 and its features into Algorithm 1, we can obtain the following API sentence: "*method get device id from class telephony manager from package Android telephony use permission read phone state throw none exception use none parameter return none*".

4.3. API sentence encoding

After completing the API sentence generation, we use the pre-trained Bert model to encode each API sentence and take the encoding output of CLS as API semantic embedding. The 768-dimensional feature vector of CLS summarizes the semantic information of the entire sentence.

Bert, a widely recognized pre-training model, has demonstrated remarkable performance across various fundamental tasks within the realm of NLP (Bengio et al., 2000). Bert adds a special word vector CLS before the input sequence. CLS is the abbreviation of classification, which can more comprehensively fuse the information of all words in the sequence. It is commonly employed in various downstream tasks, including text emotion recognition. Bert has a large number of parameters. Using Bert pre-trained on corpora such as BooksCorpus and Wikipedia (Jacob et al., 2019) can greatly save computing resources.

API sentence not only contains important features like method name and permissions of the API but also uniformly maps APIs with an inconsistent number of features into feature vectors with fixed-size. With the help of API sentence and the utilization of Bert for embedding, the extraction of semantics from an API is not solely dependent on features such as packages and parameters, but also on the method name that outlines the functionality of the API, which makes the API clustering results more accurate.

4.4. Cluster center generation

The size of the feature vector for each API, which is 768-dimensional and obtained through Bert embedding, is significantly large. This imposes a substantial burden on the distance calculation during the clustering procedure. Principal Component Analysis (PCA) (Svante et al., 1987) is employed to reduce the dimension, with a variance ratio of 90% being considered. Following the process of dimension reduction, a feature vector of 66 dimensions is obtained for each API. The Elbow method (Syakur et al., 2018) is employed to determine the optimal number of cluster centers, which is determined to be 2000. Finally, the k-Means clustering algorithm (Syakur et al., 2018) is executed.

When the Android official documentation is updated, such as adding new APIs or changing the features of existing APIs, the clustering results need to be updated. As shown in Fig. 2, We only need to generate the API sentences of newly added or changed APIs and encode them with Bert to obtain their embeddings. In the clustering process, the new API is grouped into the nearest cluster center and the coordinates of this cluster center are updated.

However, APIGRAPH (Zhang et al., 2020) needs to update the entity-relation graph and employ TransE (Bordes et al., 2013) algorithm to train the entire graph from the beginning, which generates embeddings of all APIs for re-clustering. Taking the example of updating clustering results from API level 28 (A28) to API level 29 (A29), there are 58,291 APIs in A28, and 834 new APIs have been added in A29. When updating the clustering results, our API clustering method only needs to process 834 new APIs. While APIGraph not only needs to process the 834 new APIs but also needs to process the existing 58219 APIs in A28. The API clustering method proposed in this paper has a significantly lower update cost than APIGRAPH.

5. Android malware detection

In this paper, we present AMDASE, a novel Android malware detection method that can effectively identify evolved malware. Fig. 3 illustrates the framework of Android malware detection in AMDASE.

There are four steps in Android malware detection. (1) Call Graph Extraction, uses static analysis tools to extract call graph from each app. (2) Call Graph Optimization, optimizes the call graph by deleting unknown function nodes while retaining the connectivity between their predecessor and successor nodes. (3) Feature Embedding, maps the optimized call graph into a feature vector. Initially, the function call pairs within the call graph are extracted. Subsequently, the clustering centers are employed to substitute the APIs within the call pairs, while the packages are utilized to supplant other functions. Ultimately, the feature vector is generated through the utilization of one-hot encoding. (4) Classification, uses the trained machine learning classifier to predict whether this app is benign or malicious.

5.1. Call graph extraction

The initial procedure involves employing FlowDroid (Arzt et al., 2014), a widely utilized static analysis tool, to extract the call graph from each APK file. Call graph is a tree structure that represents the functions invocation relationship of app. The Android operating system operates on an event-driven model, wherein the execution of code within an Android app is initiated by a diverse range of events (e.g., touching screen with a finger, receiving network messages). Different from common Java and C++ codes which have a clear execution order and start from a special *main()* function, the codes in Android apps have different execution orders according to the order of triggering events. Therefore, FlowDroid generates a special function named *dummyMainMethod()* as the root node of the call graph, which means the dummy

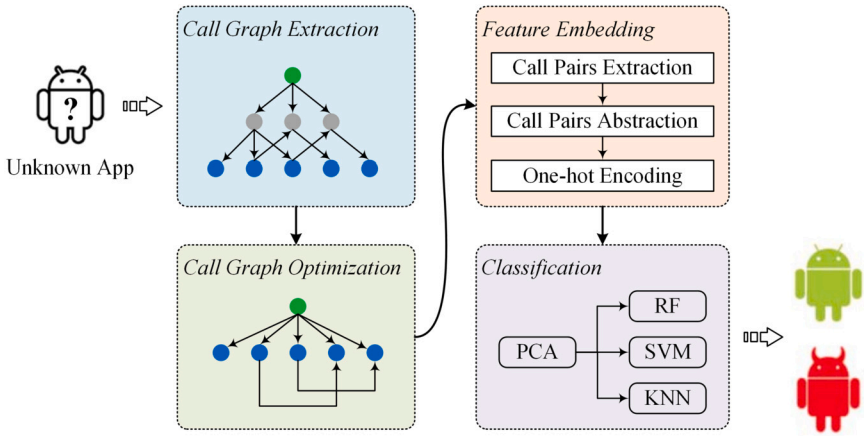

Fig. 3. Architecture of Android malware detection in AMDASE.

main function (also called entry function). The child nodes of *dummyMainMethod()* are codes corresponding to different trigger events. FlowDroid assumes that these codes can be executed in any possible order.

5.2. Call graph optimization

5.2.1. Problem caused by unknown functions

The full name of API is the application programming interface, which is a special type of invocation that implements system functions such as network communication, Bluetooth utilization, camera operations, and so forth. Besides the API, there are a large number of unknown functions and a relatively small number of other functions in the app. According to the code protection mechanism (Schulz, 2012), the unknown functions can be divided into obfuscated functions (e.g., *com.a.b.c.e123*) and self-defined functions (e.g., *com.xiaomi.mipush.sdk.pushMessageHandler*). Other functions are primarily used for class initialization (e.g., *java.io.FileOutputStream.init*). The Android official documentations provide a record of the classes to which these functions belong.

Call graph, also known as control flow graph, represents the invocation relationship within an application. Unknown functions, other functions, and APIs are all a single node in the call graph. Unknown function nodes account for the vast majority of call graph. Existing detection methods have two ways to deal with unknown functions: unified abstraction and all reservation.

Unified abstraction is abstracting all unknown function nodes into a node named *Unknown*. It will seriously reduce the structural information of the call graph. Unknown function nodes account for the vast majority of the call graph, after the abstraction of all unknown nodes, both the predecessor and successor nodes of a significant number of APIs are the *Unknown* node in the call graph. Therefore, the contextual information of APIs in the call graph is severely compromised.

All reservation involves retaining all unknown function nodes in the call graph. It will generate a lot of redundant information. The number of APIs is limited and the usage of each API is clearer. In contrast, software developers have the authority to define the names and usage of unknown functions, which can be readily altered. Retaining all unknown functions in the call graph will seriously increase the computational overhead of subsequent detection work and affect the detection method to extract API contextual information.

The API contextual information is most effective in accurately representing the behavior of each app. In essence, the primary source of relevant data for malware detection lies in the local information pertaining to API invocations within a call graph. However, the large amount of unknown function nodes greatly crippled the extraction of API contextual information.

5.2.2. Algorithm of call graph optimization

In order to solve the serious difficulty in extracting API contextual information, we propose a call graph optimization method. This approach removes nodes of unknown functions that can hardly extract any useful information while preserving the connectivity between their predecessor and successor nodes. If there exists a path consisting of all unknown function nodes between any two API nodes (e.g., APIx and APIy) in the call graph, that is, APIx calls APIy by calling several unknown functions. In the optimized call graph, APIx directly calls APIy. The unknown function itself does not contain any information, or it contains a small amount of information that can hardly be extracted. However, its invocation relationship should be preserved because it reflects the behavior of app. There are only three types of functions in the optimized call graph: entry function, API, and other functions. The invocation relationship between these three types of functions reflects the behavior of app more accurately and robustly.

Algorithm 3 shows the pseudo-code of call graph optimization. The input of this algorithm is the call graph extracted from an app and Android official documentations. The output is the optimized call graph.

Algorithm 3 Call Graph Optimization.

Input: Call Graph $CG = (V, E)$, API Documents $AODs$
1: **for** each node n in nodes **do**
2: **if** n is $entry\ function$ **then**
3: **continue**
4: **end if**
5: **if** n in $AODs$ or $n.class$ in $AODs$ **then**
6: **continue**
7: **end if**
8: collect predecessor nodes of n as $callers$
9: collect successor nodes of n as $callees$
10: **for** each node $start$ in nodes $callers$ **do**
11: **for** each node end in nodes $callees$ **do**
12: add edge $(start, end)$ to CG
13: **end for**
14: **end for**
15: remove node n from CG
16: **end for**
17: **return** CG

Lines 2 to 7 are used to filter the critical function nodes in the graph, including entry functions, APIs, and other functions.

Lines 8 to 9 are used to obtain the predecessor nodes list and the successor nodes list of the unidentified node that is intended for removal.

Lines 10 to 14 serve the purpose of maintaining the connectivity between the predecessor and successor nodes of the unknown node by nested traversing these two lists and adding a directed edge composed of any two nodes each from these two lists respectively in the call graph.

Line 15 is used to remove the unknown node from the call graph.

Table 3
Summary of the datasets used in our experiments.

App	2012	2013	2014	2015	2016	2017	2018	All
Malware (M)	4624	6519	6409	6494	6449	5388	6567	42450
Benign (B)	4418	6488	6364	6506	6474	5348	6556	42154
M + B	9042	13007	12773	13000	12923	10736	13123	84604
M / (M + B)	51.1%	50.1%	50.2%	50.0%	49.9%	50.2%	50.0%	50.2%

5.3. Feature embedding

The third step is to map the optimized call graph into a feature vector. First, we extract function call pairs from the optimized call graph, including callers and callees. The functions in call pairs can be divided into three categories: API, entry function, and other functions. Then, each API is abstracted into a cluster center obtained in Section 4. The process also involved the abstraction of other functions into their respective packages, with the entry function being designated as reserved. By abstracting APIs into the cluster centers, AMDASE is resilient to the API changes in both Android framework and malware. Finally, one-hot encoding is used to generate a feature vector of each app. The feature values of cluster call pairs in the optimized call graph are mapped to component one while the other components are set to zero.

5.4. Classification

Before classification, the dimension of the feature vector generated in the previous step is excessively large, resulting in a significant computational burden during training and prediction processes. Therefore, PCA (Svante et al., 1987) is used to reduce the dimension of the feature vector before classification. PCA can eliminate misleading information such as noise in the feature space so that the classifier can capture more stable and robust features. When testing malware developed over time, PCA can greatly improve the performance of the classifier.

The last step in Android malware detection is to perform classification using a machine learning classifier to predict whether these apps are benign or malicious. We select k-Nearest Neighbor (KNN) (Fix and Hodges, 1952), Support Vector Machines (SVM) (Hearst et al., 1998), and Random Forests (RF) (Breiman, 2001) as classifiers. These three algorithms are implemented utilizing the Python library scikit-learn (Scikit-learn, 2023). Each model is trained by the feature vectors obtained from the apps in the training set and tested by the apps from the testing set. We use 10-fold cross-validations to evaluate AMDASE.

6. Experiment and analysis

In this section, we perform a comprehensive evaluation of AMDASE, including the following six experiments.

1. We evaluate the effectiveness of AMDASE on malware detection.
2. We examine the ability of AMDASE to detect evolved Android malware.
3. We access the impact of method name in API clustering.
4. We measure the impact of call graph optimization.
5. We present the runtime evaluation of AMDASE.
6. We evaluate the ability of API clustering on API semantics extraction by analyzing the closeness of frequently used APIs in embedding space.

6.1. Experiments setup

Table 3 presents the summary of the datasets used in our experiment, including 42,450 malicious apps and 42,154 benign apps. The MD5 of apk we used was collected by the authors of APIGRAPH (Zhang et al., 2020). We downloaded malware from three open repositories, including AndroZoo (Allix et al., 2016), VirusTotal (Virustotal, 2023), and the AMD dataset (Wei et al., 2017). All the malicious apps are marked as malicious by at least 15 antivirus engines in VirusTotal. We collect benign apps from Google Play Store (2023) and download them with the help of AndroZoo. All the benign apps are marked as benign by every antivirus engine in VirusTotal.

In order to evaluate AMDASE, we perform 10-fold cross-validations using the collected datasets. We designed the following two experimental scenarios for the evaluation of evolved malware detection:

- **Scenario A**. Using apps from 2012 to 2013 for ten-fold cross-validation, which produces 10 trained classifiers. We use the 10 classifiers to detect malware developed from 2014 to 2018 and take the average of detection results.
- **Scenario B**. Using apps from 2013 to 2014 for ten-fold cross-validation, which produces 10 trained classifiers. We use the 10 classifiers to detect malware developed from 2015 to 2018 and take the average of detection results.

We choose the following widely used metrics including *Accuracy*, *Precision*, *Recall*, and *F1*-Measure. $Accuracy = (TP + TN)/(TP + TN + FP + FN)$. In the following equation, $Precision = TP/(TP + FP)$ and $Recall = TP/(TP + FN)$. *TP* and *TN*, respectively, donate the number of samples correctly classified as Malicious and Benign, while *FP* and *FN*, respectively, donate the number of samples mistakenly identified as malicious and benign.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

6.2. Malware detection performance

To evaluate the effectiveness of AMDASE on malware detection, we trained and tested AMDASE using samples developed in the same year. We conduct 10-fold cross-validations on each of the seven datasets depicted in Table 3. Table 4 presents the detection performance achieved by different methods on each dataset, respectively.

Among SVM, RF, and KNN, AMDASE achieves the best performance with RF. The average *F1*-Measure of AMDASE in 7 years has reached 96.7%, which is 4.3% and 6.4% higher than MAMADROID (92.7%) and MALSCAN (90.9%), respectively; The average *Accuracy* of AMDASE is 96.7%, which is 4.3% and 6.1% higher than MAMADROID and MALSCAN respectively; The average *Precision* of AMDASE is 98.5%, which is 5.5% and 5.6% higher than MAMADROID and MALSCAN respectively; The average *Recall* of AMDASE is 94.8%, which is 3.2% and 6.2% higher than MAMADROID and MALSCAN respectively;

It can be seen from Table 4 that all the metrics of MAMADROID increase slowly over time. MALSCAN (Wu et al., 2019) has higher *Precision* and lower *Recall* between 2012 and 2014, but lower *Precision* and higher *Recall* between 2015 and 2018. Such phenomena may be related to changes in the Android framework, but AMDASE can always maintain its detection performance stably at a high level, which is less affected by time changes.

6.3. Evolved malware detection performance

To examine how well the performance of AMDASE on evolved malware detection, we choose MAMADROID, AE-MAMADROID (Zhang et al., 2020), MALSCAN, and AE-MALSCAN (Zhang et al., 2020; Wu et

Table 4
Detection performance in the same time period.

Dataset	[Accuracy, Precision, Recall, *F1*-Measure] (%)											
	MAMADROID				MALSCAN				AMDASE			
2012	89.6	91.8	87.5	89.6	92.1	99.4	85.0	91.6	96.8	99.5	94.2	96.8
2013	89.2	90.2	88.1	89.1	88.6	98.0	78.8	87.4	95.8	97.0	94.6	95.8
2014	91.6	91.5	91.7	91.6	90.3	97.6	82.7	89.5	96.5	98.4	94.6	96.5
2015	93.2	93.9	92.4	93.2	91.6	89.1	94.8	91.9	96.5	98.7	94.1	96.4
2016	94.5	94.8	94.1	94.4	90.9	87.9	95.0	91.3	96.4	98.7	93.9	96.3
2017	94.7	95.6	93.8	94.7	91.4	90.2	92.9	91.5	97.0	98.7	95.3	97.0
2018	96.0	96.3	95.6	96.0	93.0	90.6	96.1	93.2	97.8	98.7	97.0	97.8
Average	92.7	93.4	91.9	92.7	91.1	93.3	89.3	90.9	**96.7**	**98.5**	**94.8**	**96.7**

Table 5
The *F1*-Measure (%) of five methods to perform evolved malware detection.

Scenario		2012	2013	2014	2015	2016	2017	2018	Average
Scenario A	Method	Cross-validation		Evolved malware detection					
	MAMADROID	89.9		74.2	65.8	64.4	75.9	59.1	67.9
	MALSCAN	88.1		74.0	51.0	44.7	24.0	40.1	46.8
	AE-MAMADROID	93.4		75.6	54.6	51.1	73.0	55.7	62.0
	AE-MALSCAN	91.6		78.3	59.5	52.5	31.6	41.8	52.7
	AMDASE	95.0		89.5	84.1	77.6	83.6	78.0	**82.6**
Scenario B	Method	_	Cross-validation		Evolved malware detection				
	MAMADROID	_	90.5		83.7	77.2	84.5	74.0	79.9
	MALSCAN	_	87.3		66.8	57.6	36.3	44.9	51.4
	AE-MAMADROID	_	93.3		79.7	63.3	76.8	64.5	71.1
	AE-MALSCAN	_	91.4		74.9	63.9	39.2	46.9	56.2
	AMDASE	_	94.8		90.3	84.3	88.7	82.7	**86.5**

al., 2019) as comparison methods. AE-MAMADROID and AE-MALSCAN represent using APIGRAPH to enhance MAMADROID and MALSCAN, respectively.

Among different classifiers, AMDASE has the best performance on evolved malware detection using SVM. As shown in Table 5, the *F1*-Measure of AMDASE is significantly higher than other methods. In Scenario A, the average *F1*-Measure of AMDASE from 2014 to 2018 reached 82.6%, which is 22% higher than MAMADROID (67.9%). In Scenario B, the *F1*-Measure of AMDASE from 2015 to 2018 were all higher than 80%. The experimental result proves that AMDASE trained with old samples can effectively detect evolved malware.

Compared with the API clustering method of MAMADROID, which simply abstracts API into its package, AMDASE fully considers the rich semantic information contained in the method name and permissions of API, which makes the clustering result more efficient. In the malware detection stage, AMDASE can extract more robust features reflecting app behavior with the help of call graph optimization. So that AMDASE trained by old software samples can effectively detect evolved malware and has a significantly slower aging speed.

APIGRAPH is an enhancement for existing Android malware classifiers. It makes detection methods resilient to the evolved malware by replacing APIs in the feature vectors with cluster centers obtained in APIGRAPH. MALSCAN detects malware by using the centrality information of sensitive APIs in the call graph. The sensitive APIs are collected by PScout (Au et al., 2012) and most of them are rarely used after 2016, so the *F1*-Measure of MALSCAN drops sharply in 2017. AE-MALSCAN uses the centrality information of cluster centers to replace sensitive APIs. After MALSCAN is enhanced by APIGRAPH, the aging speed slows down. On the contrary, after MAMADROID is enhanced by APIGRAPH, the aging speed becomes more serious. The specific reasons will be further analyzed in Section 6.5.

6.4. Impact of method name in API clustering

The method name of API sketches the function of API and is a vital feature of API semantics extraction. In order to access the impact of method name on the clustering results of API, we designed three malware detection methods based on API clustering. The first detection method is AMDASE with one-hot encoding (ASEOH), which uses the API clustering method proposed in this paper; The second detection method is Without Method Name (WMN). The API clustering method of WMN ignores method name when generating API sentences, and other steps are the same as the API clustering method proposed in this paper; The third detection method is APIGraph with one-hot encoding (AGOH), which employs the clustering method proposed in APIGRAPH.

These three detection methods all use the one-hot encoding of cluster centers for feature vector generation and all select the best performing classifier from KNN, RF, and SVM for malware detection. We use Scenario A and Scenario B for ten-fold cross-validation. Table 6 shows the detection results of these three detection methods based on API clustering.

In Scenario A, the average *F1*-Measure of WMN from 2014 to 2018 is 58.2%, which is 19.5% higher than AGOH; the average *F1*-Measure of ASEOH is 74.6%, which is 28.2% higher than WMN; the average *F1*-Measure of AMDASE is 82.6%, which is 10.7% higher than ASEOH. In Scenario B, the average *F1*-Measure of WMN from 2014 to 2018 is 66.9%, which is 13.4% higher than AGOH; the average *F1*-Measure of ASEOH is 79.0%, which is 18.1% higher than WMN; the average *F1*-Measure of AMDASE is 86.5%, which is 9.5% higher than ASEOH.

Both AGOH and WMN do not take method name into account while extracting API semantics, so the performance of these two methods decays quickly. Compared with AGOH, WMN uses Bert to mine semantic information in features such as class name and permissions, which results in a relatively slower aging speed. The API clustering method used by ASEOH makes full use of the semantic information contained in method name so that the performance of ASEOH is greatly improved compared with AGOH and WMN. When we use the one-hot encoding of cluster call pairs extracted from the optimized call graph as feature vector (AMDASE), the detection performance is further improved. The experiment result shows that method name plays an indispensable role in API clustering.

Table 6
The *F1*-Measure (%) of ASEOH, WMN, and AGOH to perform evolved malware detection.

Scenario		2012	2013	2014	2015	2016	2017	2018	Average
Scenario A	Method	Cross-validation		Evolved malware detection					
	AGOH	91.9		73.4	55.7	49.2	29.3	36.1	48.7
	WMN	94.7		81.9	59.9	51.4	40.3	57.4	58.2
	ASEOH	95.7		86.7	68.1	68.7	79.5	70.0	**74.6**
Scenario B	Method	_	Cross-validation		Evolved malware detection				
	AGOH	_	92.4		75.0	67.2	42.4	51.5	59.0
	WMN	_	92.8		78.5	73.5	51.6	64.1	66.9
	ASEOH	_	91.8		81.7	77.6	83.3	73.2	**79.0**

Table 7
The *F1*-Measure (%) of AMDASE with and without call graph optimization to perform evolved malware detection.

Scenario		2012	2013	2014	2015	2016	2017	2018	Average
Scenario A	Method	Cross-validation		Evolved malware detection					
	AWOCGOPT	90.7		73.2	55.4	58.4	32.4	40.8	52.0
	AMDASE	95.0		89.5	84.1	77.6	83.6	78.0	**82.6**
Scenario B	Method	_	Cross-validation		Evolved malware detection				
	AWOCGOPT	_	91.2		75.6	73.5	61.2	62.5	68.1
	AMDASE	_	94.8		90.3	84.3	88.7	82.7	**86.5**

Table 8
Number and proportion of key features in feature space of AMDASE, MAMADROID, and AE-MAMADROID.

Dataset	MAMADROID	AE-MAMADROID	AMDASE
2012	65 (23.9%)	149 (5.0%)	4183 (69.2%)
2013	106 (30.59%)	223 (6.6%)	4333 (66.2%)
2014	200 (41.1%)	295 (7.3%)	7488 (65.7%)
2015	285 (44.1%)	415 (8.9%)	8363 (62.1%)
2016	435 (47.5%)	466 (8.6%)	9488 (53.1%)
2017	376 (45.5%)	429 (8.8%)	8897 (51.8%)
2018	745 (57.2%)	627 (9.6%)	11043 (47.2%)
Average	316 (41.4%)	372 (7.8%)	**7685 (57.7%)**

6.5. Impact of call graph optimization

Call graph optimization removes all unknown function nodes in the call graph while preserving the connectivity between their predecessor and successor nodes, which enables AMDASE to extract more robust features that reflect the behavior of app. In order to measure the impact of call graph optimization on the aging speed of AMDASE, we compare the performance of AMDASE with and AMDASE without call graph optimization (AWOCGOPT). Scenario A and Scenario B were selected for ten-fold cross-validation. Table 7 shows the detection performance of AMDASE with and without call graph optimization.

It can be seen that call graph optimization only slightly improves the detection performance of AMDASE in cross-validation, but it can significantly improve the performance of evolved malware detection. In Scenario A, AMDASE has an average *F1*-measure of 82.6% from 2014 to 2018 with the step of call graph optimization, which is 59% higher than that without call graph optimization (52.0%). In Scenario B, AMDASE has an average *F1*-Measure of 86.7% from 2015 to 2018 with the step of call graph optimization, which is 27% higher than that without call graph optimization (68.1%).

In Section 6.3, APIGRAPH weakens the ability of MAMADROID on evolved malware detection, We believe that this is due to the lack of call graph optimizations, resulting in a large number of unknown function nodes in the call graph crippled the extraction of API contextual information. Compared with unknown functions, cluster call pairs are key features that can better reflect the behavior of app and can make the detection method resilient to the frequent changes of API in malware. Table 8 presents the number and proportion of key features in feature vectors of MAMADROID AE-MAMADROID, and AMDASE.

Table 8 shows that package call pairs in MAMADROID only account for 41.4% of all features on average. MAMADROID abstracts API into its package and uses unified abstraction to deal with the unknown functions rather than call graph optimization. Unknown functions are abstracted into *self-defined* or *obfuscated*, resulting in nearly 60% of the features being mainly invocations between packages and *self-defined* methods.

AE-MAMADROID replaces the APIs in call pairs with cluster centers and other steps are the same as MAMADROID. As shown in Table 8, after 'enhanced' by APIGRAPH, only 372 cluster call pairs can be extracted each year on average, accounting for 7.8% of the total number of features. More than 92% of the features are invocations between cluster centers, *self-defined* methods, and *obfuscated* methods. The huge decrease in the proportion of key features causes APIGRAPH to accelerate the aging speed of MAMADROID.

On the other hand, AMDASE can extract 7685 cluster center call pairs each year on average, accounting for 57.7% of the features after call graph optimization. This shows that call graph optimization enables the detection method to extract more critical features that represent the mode of app behavior, which is indispensable for evolved malware detection.

6.6. Runtime evaluation

The device used in runtime evaluation is a MacBook Pro laptop equipped with M1 Pro (8×3.2GHz) chip and 16GB memory. In this section, we present the runtime overhead of MAMADROID, MALSCAN, and AMDASE by using 2000 apps randomly selected from the datasets listed in Table 3. Table 9 presents the result of the runtime evaluation.

Same as AMDASE, both MAMADROID and MALSCAN are malware detection methods using API contextual information in the call graph. Therefore, the runtime of these three methods can be divided into three phases: Call Graph Extraction, Feature Extraction, and Classification.

- **Call Graph Extraction.** This phase extracts call graph from the APK file. Both MAMADROID and AMDASE use Soot (Vallée-Rai et al., 2010) for code representation and use FlowDroid to generate call graph, so the time cost of these two methods are almost the same (14.345s and 14.297s). MALSCAN uses Androguard (Androguard, 2023) to generate call graphs. Compared with FlowDroid, the time cost of Androguard is greatly reduced (3.912s, about 72.6% lower).

Table 9
Runtime evaluation (seconds).

Method	Call Graph Extraction	Feature Extraction	Classification	Total
MAMADROID	14.345	0.529	0.028	14.902
MALSCAN	3.912	3.293	0.003	**7.208**
AMDASE	14.297	0.974	0.025	15.296

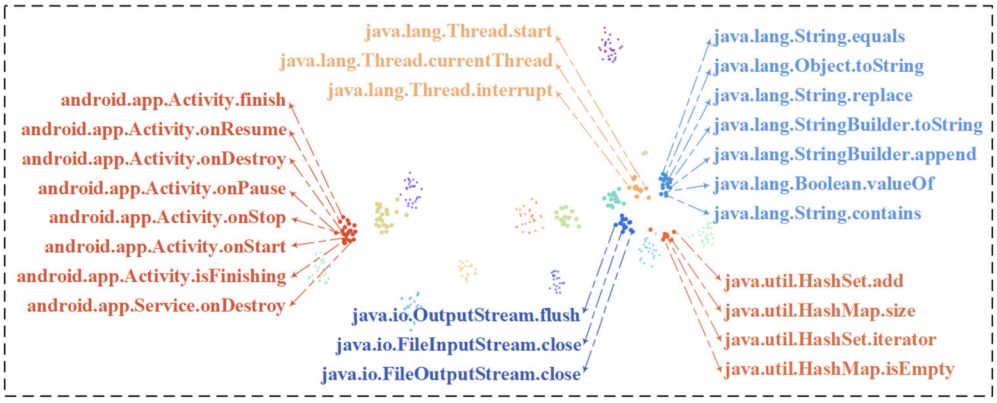

Fig. 4. Visualization of frequently used APIs in embedding space. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

- **Feature Extraction.** This phase converts the call graph into a feature vector. MALSCAN needs to calculate the centrality information of sensitive APIs in the call graph, the time cost is the largest (3.293s). AMDASE needs to optimize the call graph and embed features at this phase. The time cost is 0.974s. MAMADROID abstracts function call pairs from the call graph into package call pairs and constructs Markov chains to model the transition probabilities, which takes the least time (0.529s).
- **Classification.** This phase uses a trained machine learning classifier to predict whether the app is malicious or benign. MAMADROID uses PCA for dimension reduction and RF for detection. The time cost of MAMADROID is 0.028s. AMDASE uses PCA for dimension reduction and SVM for detection. The time cost of AMDASE is 0.025s. MALSCAN only needs RF for detection in this phase and the time overhead is the least (0.003s).

MALSCAN focuses on quick scanning of a large number of apps in app stores (such as Google Play). Therefore, the time required in MALSCAN is the least (7.208s) among these three methods. The runtime overhead of MAMADROID and AMDASE is relatively large (14.902s and 15.296s). Among them, AMDASE has the longest runtime, but considering the excellent detection performance and extremely slower aging speed of AMDASE, the runtime performance of AMDASE is acceptable.

6.7. Ability of API clustering on semantics extraction

In this section, we evaluate the ability of API clustering on API semantics extraction by analyzing the closeness of frequently used APIs in the embedding space. We selected the most frequently used 200 APIs and used t-SNE (Van der Maaten and Hinton, 2008) algorithm to map the high-dimensional feature vectors in the embedding space to a two-dimensional plane. Due to space limitations, Fig. 4 only shows a part of the visualization results.

In Fig. 4, APIs belonging to the same cluster center are marked with the same color. Red nodes are APIs related to the lifecycle of Android activities, yellow nodes are APIs related to activities of thread, dark blue nodes are APIs related to input and output, orange nodes are APIs related to hash maps, and light blue nodes are APIs related to operations of array content. It can be seen that APIs belonging to the same cluster center have similar functions, while APIs with different functions are grouped into different cluster centers.

It is worth noting that both the light blue nodes and the yellow nodes are APIs that belong to the Android package *java.lang*. This suggests that the clustering method used by MAMADROID, which simply groups APIs into their packages, is insufficient to extract API semantics.

As shown in Fig. 4, the Euclidean distance between APIs with similiar functionality is shorter. This demonstrates the effectiveness of our API clustering method in semantics extraction. Our method can generate embeddings more accurately reflect the functionality of API, which makes the subsequent k-Means clustering algorithm to better group APIs with similar functions into the same cluster center.

7. Discussion

Rule of API Sentence. When designing the rules of API sentence generation, we found that although some rules with different expressions have the same meaning, the API clustering results can have a certain impact on the detection performance of AMDASE. For example, changing the rule "*use none permission*" to "*do not use permission*" or "*use no permission*" will weaken the ability of AMDASE to detect evolved malware. Perhaps Bert is able to extract more information from the word 'none' than 'not' or 'no'.

API Clustering. API clustering belongs to unsupervised learning. There is no absolutely correct answer to judge the quality of API clustering results. The effectiveness of clustering can only be reflected through the performance of malware detection. The API clustering method proposed in this paper can effectively delay the aging speed of the classifier, which makes the classifier resilient to the API changes in Android framework.

Obfuscation Techniques. Malware may use obfuscation techniques such as packaging (Duan et al., 2018; Lu et al., 2020, 2021a), reflection, and dynamic code loading (Falsina et al., 2015) to evade static detection. Malware developers may obfuscate API as unknown functions. During the call graph optimization, these obfuscated APIs are lost. However, the intention of call graph optimization is that we cannot find an efficient way to obtain useful information from a large number of unknown functions. Compared with benign app, malware usually adds more API call pairs during call graph optimization. These call pairs are mostly related to thread activity. Malware often masquerades

as benign app and performs malicious actions through multi-threaded activity.

Deep Learning Classifier. Deep learning has been widely used in network security. In the future, we will try to use DNN or GNN (Wu et al., 2020) as classifiers for malware detection. This paper believes that simple machine learning classifiers (such as SVM, and KNN) are more suitable for detecting concept drift samples, and stronger classifiers (such as random forests) are prone to cause an over-fitting phenomenon. Based on their own characteristics, these complex algorithms can learn more correlations other than malicious behaviors in the training phase and achieve better detection results, which also makes them difficult to detect evolved malware effectively.

Runtime. It should be noticed that in Section 6.6, the time cost of MAMADROID and AMDASE in the first phase accounts for about 95% of the total time. Using Androguard to extract call graph can greatly reduce the time overhead of malware detection, but at the same time, a certain amount of call graph accuracy will be lost, which will eventually affect the detection performance. How to quickly and accurately extract the call graph of Android apps is still an important research direction in the future.

8. Conclusion

Detecting continuously evolving malware is a significant challenge. To address this issue, we propose AMDASE, a novel Android malware detection method with API semantics extraction. We design API sentence to summarize important API features and use Bert to extract the semantics of API. With the help of API sentence, the extraction of semantic information from an API is not solely dependent on features such as packages and parameters, but also on the method name that outlines the functionality of the API. In order to overcome the difficulty in extracting API contextual information caused by a large number of unknown functions in the call graph, we propose a call graph optimization approach that removes all unknown nodes in the call graph while maintaining the connection between their predecessor and successor nodes. Meanwhile, we abstract the APIs in the optimized call graph into API cluster centers that represent their functions, which makes the detection method resilient to the frequent API changes in both Android framework and malware. We evaluate the effectiveness of AMDASE on a dataset of 42.2K benign and 42.5K malicious apps developed over seven years. The experimental results show that AMDASE greatly outperforms the existing start-of-the-art Android malware detection methods, while also exhibiting a noticeably reduced rate of aging.

CRediT authorship contribution statement

Hongyu Yang: Conceptualization, Funding acquisition, Writing – review & editing. **Youwei Wang:** Conceptualization, Investigation, Methodology, Software, Writing – original draft, Writing – review & editing. **Liang Zhang:** Conceptualization, Writing – review & editing. **Xiang Cheng:** Conceptualization, Supervision. **Ze Hu:** Conceptualization, Funding acquisition, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgement

This research was supported by the National Natural Science Foundation of China (Grant No. 62201576, U1833107), the Scientific Research Project of the Tianjin Municipal Education Commission (Grant No. 2019KJ127), the Supporting Fund Project of the National Natural Science Foundation of China (Grant No. 3122023PT10), and the Discipline Development Funds of Civil Aviation University of China.

References

Allen, J., Landen, M., Chaba, S., Ji, Y., Chung, S.P.H., Lee, W., 2018. Improving accuracy of Android malware detection with lightweight contextual awareness. In: Proceedings of the 34th Annual Computer Security Applications Conference. Association for Computing Machinery, pp. 210–221.

Allix, K., Bissyandé, T.F., Klein, J., Le Traon Androzoo, Y., 2016. Collecting millions of Android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories. Association for Computing Machinery, pp. 468–471.

Androguard, 2023. https://github.com/androguard/androguard.

Api documentations, 2023. https://developer.android.com/reference.

Apigraph, 2023. https://github.com/seclab-fudan/APIGraph.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C., 2014. Drebin: effective and explainable detection of Android malware in your pocket. In: Proceedings of the 2018 Network and Distributed Systems Security Symposium.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. ACM SIGPLAN Not. 49, 259–269.

Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D., 2012. Pscout: analyzing the Android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. Association for Computing Machinery, pp. 217–228.

Bengio, Y., Ducharme, R., Vincent, P., 2000. A neural probabilistic language model. Adv. Neural Inf. Process. Syst. 13.

Bhat, P., Behal, S., Dutta, K., 2023. A system call-based Android malware detection approach with homogeneous & heterogeneous ensemble machine learning. Comput. Secur. 130, 103–277.

Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., Yakhnenko, O., 2013. Translating embeddings for modeling multi-relational data. Adv. Neural Inf. Process. Syst. 26.

Breiman, L., 2001. Random forests. Mach. Learn. 45, 5–32.

Chen, L., Hou, S., Ye, Y., Xu Droideye, S., 2018. Fortifying security of learning-based classifier against adversarial Android malware attacks. In: Proceedings of the 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining. IEEE, pp. 782–789.

Duan, Y., Zhang, M., Bhaskar, A.V., Yin, H., Pan, X., Li, T., Wang, X., Wang, X., 2018. Things you may not know about Android (un) packers: a systematic study based on whole-system emulation. In: Proceedings of the 2018 Network and Distributed Systems Security Symposium.

Falsina, L., Fratantonio, Y., Zanero, S., Kruegel, C., Vigna, G., Maggi, F., 2015. Grab'n run: secure and practical dynamic code loading for Android applications. In: Proceedings of the 31st Annual Computer Security Applications Conference. Association for Computing Machinery, pp. 201–210.

Feng, R., Chen, S., Xie, X., Meng, G., Lin, S., Liu, Y., 2020. A performance-sensitive malware detection system using deep learning on mobile devices. IEEE Trans. Inf. Forensics Secur. 16, 1563–1578.

Fix, E., Hodges, J.L., 1952. Discriminatory analysis. Nonparametric discrimination: Small sample performance.

Google play store. https://play.google.com/store.

Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P., 2017. Adversarial examples for malware detection. In: Proceedings of the Computer Security – ESORICS 2017. Springer, pp. 62–79.

Gu, Y., Li, L., 2021. Graphevolvedroid: mitigate model degradation in the scenario of Android ecosystem evolution. In: Proceedings of the 30th ACM International Conference on Information & Knowledge Management. Association for Computing Machinery, pp. 3588–3591.

Hearst, M.A., Dumais, S.T., Osuna, E., Platt, J., Scholkopf, B., 1998. Support vector machines. IEEE Intell. Syst. Appl. 13, 18–28.

Hei, Y., Yang, R., Peng, H., Wang, L., Xu, X., Liu, J., Liu, H., Xu, J., Sun Hawk, L., 2021. Rapid Android malware detection through heterogeneous graph attention networks. IEEE Trans. Neural Netw. Learn. Syst., 1–15.

Jacob, D., Ming-Wei, C., Kenton, L., Toutanova Bert, K., 2019. Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 NAACL-HLT, volume 1. Association for Computational Linguistics, pp. 4171–4186.

Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L., 2017. Transcend: detecting concept drift in malware classification models. In: Proceedings of the 26th USENIX Security Symposium. USENIX Association, pp. 625–642.

Karbab, E.B., Debbabi, M., 2021. Petadroid: adaptive Android malware detection using deep learning. In: Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 319–340.

Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D., 2018. Maldozer: automatic framework for Android malware detection using deep learning. Digit. Investig. 24, S48–S59.

Lau, J.H., Baldwin, T., 2016. An empirical evaluation of doc2vec with practical insights into document embedding generation. ArXiv preprint arXiv:1607.05368.

Lei, T., Qin, Z., Wang, Z., Li, Q., Ye, D., 2019. Evedroid: event-aware Android malware detection against model degrading for iot devices. IEEE Int. Things J. 6, 6668–6680.
Liu, X., Liu, J., Zhu, S., Wang, W., Zhang, X., 2019. Privacy risk analysis and mitigation of analytics libraries in the Android ecosystem. IEEE Trans. Mob. Comput. 19, 1184–1199.
Lu, H., Jin, C., Helu, X., Zhu, C., Guizani, N., Tian, Z., 2020. Autod: intelligent blockchain application unpacking based on jni layer deception call. IEEE Netw. 35, 215–221.
Lu, H., Jin, C., Helu, X., Du, X., Guizani, M., Tian, Z., 2021a. Deepautod: research on distributed machine learning oriented scalable mobile communication security unpacking system. IEEE Trans. Netw. Sci. Eng. 9, 2052–2065.
Lu, H., Jin, C., Helu, X., Zhang, M., Sun, Y., Han, Y., Tian, Z., 2021b. Research on intelligent detection of command level stack pollution for binary program analysis. Mob. Netw. Appl. 26, 1723–1732.
Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G., 2016. Mamadroid: detecting Android malware by building Markov chains of behavioral models. ArXiv preprint arXiv:1612.04433.
Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., et al., 2019. Tesseract: eliminating experimental bias in malware classification across space and time. In: Proceedings of the 28th USENIX Security Symposium. USENIX Association, pp. 729–746.
Qiu, J., Zhang, J., Luo, W., Pan, L., Nepal, S., Xiang, Y., 2020. A survey of Android malware detection with deep neural models. ACM Comput. Surv. 53, 1–36.
Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K., 2022. Dos and don'ts of machine learning in computer security. In: Proceedings of the 31st USENIX Security Symposium. USENIX Association, Boston, MA, pp. 3971–3988.
Schulz, P., 2012. Code Protection in Android. Institute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany. 110.
Scikit-learn, 2023. https://scikit-learn.org/.
Svante, W., Kim, E., Paul, G., 1987. Principal component analysis. Chemom. Intell. Lab. Syst. 2, 37–52.
Syakur, M., Khotimah, B., Rochman, E., Satoto, B.D., 2018. Integration k-means clustering method and elbow method for identification of the best customer profile cluster. In: Proceedings of the IOP Conference Series. In: Materials Science and Engineering, vol. 336. IOP Publishing, 012017.
Tarwireyi, P., Terzoli, A., Adigun, M.O., 2023. Using multi-audio feature fusion for Android malware detection. Comput. Secur. 131, 103–282.
Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 2010. Soot: a Java bytecode optimization framework. In: CASCON First Decade High Impact Papers, pp. 214–224.
Van der Maaten, L., Hinton, G., 2008. Visualizing data using t-sne. J. Mach. Learn. Res. 9.
Virustotal, 2023. https://www.virustotal.com/gui/home/upload.
Wang, W., Li, Y., Wang, X., Liu, J., Zhang, X., 2018. Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers. Future Gener. Comput. Syst. 78, 987–994.
Wang, W., Zhao, M., Wang, J., 2019. Effective Android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J. Ambient Intell. Humaniz. Comput. 10, 3035–3043.
Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W., 2017. Deep ground truth analysis of current Android malware. In: Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 252–276.
Wu, Y., Li, X., Zou, D., Yang, W., Zhang, X., Jin Malscan, H., 2019. Fast market-wide mobile malware scanning by social-network centrality analysis. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 139–150.
Wu, Y., Li, M., Zeng, Q., Yang, T., Wang, J., Fang, Z., Cheng, L., 2023. Droidrl: feature selection for Android malware detection with reinforcement learning. Comput. Secur. 128, 103–126.
Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y., 2020. A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. 32, 4–24.
Xu, J., Li, Y., Deng, R.H., Xu, K., 2020. Sdac: a slow-aging solution for Android malware detection using semantic distance based api clustering. IEEE Trans. Dependable Secure Comput. 19, 1149–1163.
Xu, K., Li, Y., Deng, R., Chen, K., Xu, J., 2019. Droidevolver: self-evolving Android malware detection system. In: Proceedings of the 2019 IEEE European Symposium on Security and Privacy. IEEE, pp. 47–62.
Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras Droidminer, P., 2014. Automated mining and characterization of fine-grained malicious behaviors in Android applications. In: Proceedings of the Computer Security - ESORICS 2014. Springer, pp. 163–182.
Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., Enck, W., 2015. Appcontext: differentiating malicious and benign mobile app behaviors using context. In: Proceedings of the 37th IEEE International Conference on Software Engineering, volume 1. IEEE, pp. 303–313.
Yuan, C., Cai, J., Tian, D., Ma, R., Jia, X., Liu, W., 2022. Towards time evolved malware identification using two-head neural network. J. Inf. Secur. Appl. 65, 103098.
Zhang, M., Duan, Y., Yin, H., Zhao, Z., 2014. Semantics-aware Android malware classification using weighted contextual api dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, pp. 1105–1116.
Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., Yang, M., 2020. Enhancing state-of-the-art classifiers with api semantics to detect evolved Android malware. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, pp. 757–770.

Hongyu Yang received the Ph.D. degree in 2003 from Tianjin University, China in Computer Science. Since 1997, he has been with the School of Computer Science and Technology and the School of Safety Science and Engineering at Civil Aviation University of China, where he is currently a Full Professor. His research interest includes network and system security, information security, and software security.

Youwei Wang received a bachelor degree in Nanjing University of Posts and Telecommunications. He is currently a graduate student in Civil Aviation University of China. His research interests are mainly focused on Android malware detection, API semantics, and machine learning.

Liang Zhang received a Ph.D. from Tianjin University. He is currently an experimentalist in School of Information at the University of Arizona. His research interests are mainly focused on network and system security, reinforcement learning, and deep learning-based signal processing.

Xiang Cheng received the Ph.D. from Nanjing University of Aeronautics and Astronautics. He is currently an experimentalist with the school of Information Engineering, Yangzhou University, Yangzhou. His research interests are mainly focused on federated learning, network security, and advanced persistent threats.

Ze Hu received a Ph.D. from Harbin Institute of Technology. He is currently a senior lecturer in School of Safety Science and Engineering at Civil Aviation University of China. His research interests are mainly focused on medical informatics, natural language processing, and network information security.