Sealing Technology
ISSN: 1350-4789

Probabilistic Inference and Trustworthiness Evaluation of Associative Links toward Malicious Attack Detection for Online Recommendations

Vanapamula Veerabrahmachari, Arekatla Madhava Reddy, Dr. Inaganti Shylaja,

Dr. Padigala Suresh

[1,2] Assistant Professor, [3,4] Professor

vveerabrahmachari@gmail.com, amreddy2008@gmail.com

shyalajainaganti@gmail.com, padigalas36@gmail.com

Department of CSE, A M REDY MEMORIAL COLLEGE OF ENGINEERING AND TECHNOLOGY,

PETLUVARI PALEM, ANDHRA PRADESH-522601

Abstract:

Malware detection systems have faced significant hurdles and strain in recent years due to the fast growth in the quantity and variety of Android malware. Static detection is a popular technique in academics and business for identifying Android malware. However, the current static detecting approaches compromise the unduly high analysis complexity and time cost in order to enhance the detection accuracy. Furthermore, a significant quantity of data becomes redundant due to the connection between static characteristics. As a result, our research suggests a static technique based on sensitive patterns for identifying Android malware. It reduces the creation of superfluous data by mining common combinations of sensitive permissions and API requests in both dangerous and benign applications using an enhanced FP-growth algorithm. Furthermore, the multi-layered gradient boosting decision trees approach is used in this work to train the detection model. Additionally, a dual similarity combination approach is suggested to assess how similar certain sensitive patterns are to one another. The experimental findings demonstrate the excellent generalization capacity and high accuracy of our suggested detection approach.

Key words: Static detection; Sensitive pattern; Android malware

1 INTRODUCTION

The use of different mobile communication devices, such as smartphones, tablets, and wristbands, is rising in tandem with the rapid development of these technologies. Numerous mobile terminal operating systems have surfaced to provide a satisfactory user experience; the Android operating system (OS) has the most market share among them. As to the International Data Corporation (IDC) study on the worldwide smartphone operating system industry, Android holds the top position with a market share of 86.7% [1]. As shown in Fig. 1, the Android system often uses a hierarchical design architecture. The Linux kernel serves as the foundation for the Android system's kernel area. The virtual machine operating environment, API framework, and local system library make up the top user space of the Android system. Permission access requests serve as the link between the user and kernel spaces. This hierarchical structure has the benefit of hiding the particulars of the bottom layer's implementation. It may conceal layer discrepancies and provide consistent services to the top levels using the lower layers. Therefore, when changes happen at a lower layer, the Android system's hierarchical structure is unaffected on the top layer. Moreover, fixed Service Access Points (SAP) may be provided by each layer to attain high cohesiveness and low coupling. The Android system's hierarchical structure provides a robust open-sourcing feature and a relaxed approach for application release certification. With the Android system, developers are free to publish their own applications without any limitations. Additionally, users have access to a variety of download locations for these necessary apps, such as third-party app markets and official app stores. However, Android has emerged as one of the most susceptible platforms to malware attacks due to its open-sourcing characteristic.

According to McAfee's mobile threat report [2], which was published in the first quarter of 2021,

there are already over 40 million mobile malware samples in total, and the growth pace is still quite rapid. Through analyzing the malevolent actions of malware, we have ascertained that the majority of them aim to pilfer users' confidential data, including personal images, bank accounts, emails, text messages, and phone contacts. This has the negative effect of enabling criminals to gather this private data via malware and utilize it for illicit operations to generate revenue.

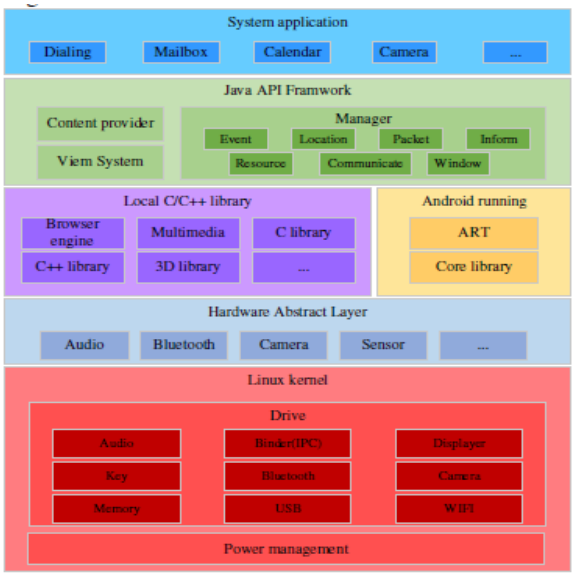

Fig. 1: Android system architecture.

Researchers in academia and business have developed several strategies and tools for detection in order to safeguard users from malware and provide a secure and healthy mobile communication environment. Static detection is a commonly used technique in the identification of Android malware. This technique examines the content of an Android application's APK files statically in order to identify any possible security risks. Almost all static detection techniques used nowadays are built on APK files. In order to provide a thorough description of data samples, existing static detection algorithms often use the concept of "comprehensive analysis," which entails parsing as many files as possible from the compressed package and extracting various sorts of feature information from them. Even while this method may provide outcomes that are somewhat good, there are additional issues.

For instance, a thorough analysis results in an excessive amount of analysis complexity and time; also, data redundancy is caused by feature correlations. Thus, the keys to figuring out if a detection technique is efficient are what to pick as the foundation for analysis and how to properly describe the data sample.

2 RELATED WORK

A significant amount of relevant work has been done to support the advancement of malware detection [3–4].

2.1 Analysis of Static Detection

The majority of conventional detection techniques rely on the signature authentication mechanism [5–6]. They entered the known harmful software's signatures into the database. After that, the sample's signature was taken out and matched to the database to see whether any dangerous software was there. Despite being straightforward and successful, this strategy has two significant flaws: First, the unknown malware cannot be found using the conventional way. Malware can evade signature-based identification by making minor code changes in an application that do not impact semantics. This is because the corresponding signature of unknown malware does not exist in the database and it is costly to create a new signature and publish it through other methods.

Malware detection now includes more static characteristics in order to address the aforementioned issues. Malware often has to request the necessary authorizations before it can carry out destructive operations like sending text messages and accessing contact lists. As a result, permission-based static detection techniques have been put out. As an example, the authors [7] compared how permissions were used by malicious and benign programs. They discovered that there was no practical way to differentiate them using those shared rights. When the amount of permissions needed was little, however, the difference became clear; Huang Liang et al. [8] created a list of all permission combinations that typically occur in malware and utilized it to create a detection model.

The API, which serves as the foundation for implementing application functionality in the Android system, is a good representation of an application's behavior. Thus, many techniques for static detection that rely on API analysis have been researched. The writers [9] disassembled APK files, for instance. It might gather API requests that pose a significant risk to user security via data stream analysis; To increase the detection accuracy, the authors in [10] examined a variety of characteristics in addition to permissions and APIs, including activities, services, intents, and network addresses; TaeGuen Kim et al. [11] expanded the feature set to include opcodes and environment variables.

Furthermore, according to some studies, the use of string characteristics is susceptible to dimensionality. However, structural characteristics are better suited for handling large amounts of data. Wei Wang et al. [13] and Yao Du et al. [14] created the function call graph based on the call relationship between methods, while Jixin Zhang et al. [12] built the Dalvik opcode graph and examined its topology features, such as node number, probability density, and graph distance, to characterize malware. In order to classify malware, graph similarity was determined.

2.2 Artificial Intelligence

The fast advancement of artificial intelligence in recent years has made machine learning a prominent topic for study across many disciplines. Its ideas and techniques have been used extensively in engineering and scientific research to overcome challenging issues. The core of malware detection is a classification problem that satisfies relevant machine learning constraints. As a result, enhancing detection efficiency using machine learning has emerged as a key area of research for malware detection.

In machine learning, supervised learning is one of the most used techniques for training models. From the labeled data, it derives the mapping relationship (i.e., function), and then uses this connection to conduct instance inferences on the unlabeled data. Support vector machines (SVM), decision trees, K closest neighbors (KNN), Bayesian networks, and others are examples of common supervised learning methods. Because labeled data gathering is rather expensive, supervised learning really takes a lot of time and highly qualified professionals to finish. Unsupervised learning has thus emerged as a different technique for overcoming this constraint. The training data in unsupervised learning is not labeled. The association between the variables is discovered by the model itself during its "self-learning" training phase. The clustering method is the most often used unsupervised learning algorithm. Literature [14], for instance, examined the clustering K-MEANS method. In order to create feature vectors, it gathered the runtime traffic of Android apps and chose six characteristics: frame length, frame number, connection duration, relative duration (time from the first frame), source port, and destination port. The experiment in [14] demonstrated the excellent malware detection accuracy of the KMEANS algorithm.

Deep learning has been steadily used in malware detection with the arrival of neural networks and the big data age. In reference [15], the writers established a correlation between the attributes of static and dynamic analysis. In [16], the authors used system calls and permissions to model neural networks; in [17], the authors proposed a convolutional neural network-based system for Android malware detection that used the application's original opcode sequence as a feature; in [18], the detection system employed a variety of classifiers, including deep neural networks. Deep belief networks were used to characterize malware in this study. It made a variety of data entry possible, including permissions, intents, system actions, and API calls.

3 PROPOSED METHOD

This research uses machine learning to provide a static detection approach based on sensitive pattern for Android malware detection. The general design of the suggested technique is shown in Fig. 2. To find out about permissions and API calls, it first disassembles the Android applications' APK files. Subsequently, sensitive patterns of both malware and regular software are produced by clustering patterns, mining common combinations, and filtering raw data. The feature vectors containing the sensitive patterns are then built.

Lastly, a machine learning approach is used to train the malware detection model.

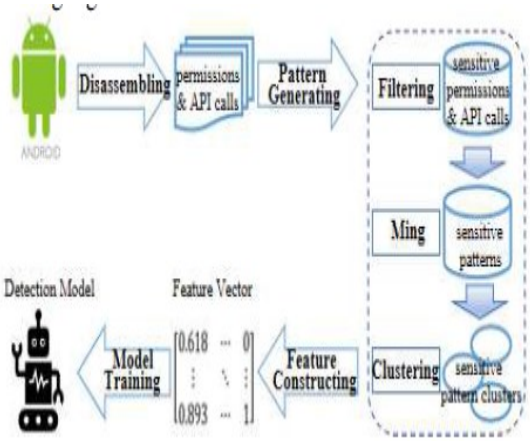

Fig. 2: The overall architecture of the method.

3.1 Extraction of Raw data

In our suggested approach, sensitive patterns cannot be created until the Android software's raw data has been retrieved. Reverse engineering tools must thus be used in order to retrieve the permission and APK files from the raw data. As shown in Tab. 1, the reverse engineering tool used in this work is Apktool [19–20]. It offers a ton of commands for APK file compilation and decompilation. To begin the decompilation of the APK file, enter "apktool d xx.apk" on the command line window, where xx is the name of the APK file. After the command is

executed, a folder containing different decompiled files will be generated in the same directory as the APK file. The AndroidManifest.xml file, smali file, res file, and assets file are the primary files in the folder. The suggested approach focuses on the permissions data from the androidmanifest.xml file and the API call data from the smali file.

Tab 1: The relevant Apktool instructions (partial).

apktool d[ecode][options] <file_apk>		
-f	--force	Force delete destination directory
-o	--output <dir>	The name of folder that gets written
-p	--frame-path <dir>	Uses framework files located in <dir>

-r	--no-res	Do not decode resources
-s	--no-src	Do not decode source
-t	--frame-tag <tag>	Uses framework files tagged by <tag>
-r	--force-all	Skip changes detection and build all files
-o	--output <dir>	The name or apk that gets written
-p	--frame-path <dir>	Uses framework files located in <dir>

Android offers an application framework with a permission-based security paradigm to limit application access to permissions including phone, network, contacts, SMS, and GPS position. As shown in Fig. 3, the developer must utilize the <uses-permission> element in AndroidManifest.xml to specify the necessary permissions. Normal permission, hazardous permission, and signing permission are the three categories into which these permissions are separated [21]. By matching the term ".permission" in the AndroidManifest.xml file, our suggested technique obtains the permission information.

(1) Normal: The least dangerous kind of authorization for the user, system program, or device is this one. When an app is installed, regular rights are often provided by default.

(2) Risky: This kind of authorization may access the device's critical sensors and personal info.

As a result, when an application is installed, users voluntarily provide potentially harmful rights.

(3) Signing: System programs have access to signing rights. When an app requests permission and is signed with the same developer certificate as the app that requested it, the permission is granted.

```
4    <uses-permission android:name="android.permission.CAMERA"/>
5    <uses-permission android:name="android.permission.FLASHLIGHT"/>
6    <uses-permission android:name="android.permission.INTERACT_ACROSS_USERS"/>
7    <uses-permission android:name="android.permission.INTERACT_ACROSS_USERS_FULL"/>
8    <uses-permission android:name="android.permission.INTERNET"/>
9    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
10   <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
11   <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
12   <uses-permission android:name="android.permission.CHANGE_NETWORK_STATE"/>
13   <uses-permission android:name="android.permission.MODIFY_PHONE_STATE"/>
14   <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
15   <uses-permission android:name="android.permission.ACCESS_DOWNLOAD_MANAGER"/>
16   <uses-permission android:name="android.permission.ACCESS_CACHE_FILESYSTEM"/>
17   <uses-permission android:name="android.permission.SEND_DOWNLOAD_COMPLETED_INTENTS"/>
18   <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
19   <uses-permission android:name="android.permission.ACCESS_ALL_DOWNLOADS"/>
20   <uses-permission android:name="android.permission.UPDATE_DEVICE_STATS"/>
21   <uses-permission android:name="android.permission.CONNECTIVITY_INTERNAL"/>
22   <uses-permission android:name="android.permission.MODIFY_NETWORK_ACCOUNTING"/>
23   <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
24   <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
25   <uses-permission android:name="android.permission.WAKE_LOCK"/>
```

Fig. 3: Permissions declared in AndroidManifest.xml

```
.method public answerRingingCall()V
    .locals 1

    .prologue
    .line 212
    invoke-virtual {p0}, Laa;->a()Z

    .line 213
    iget-object v0, p0, Laa;->c:Lcom/android/internal/telephony/ITelephony;

    invoke-interface {v0}, Lcom/android/internal/telephony/ITelephony;->answerRingingCall()V

    .line 214
    return-void
.end method

.method public dailPhone(Landroid/content/Context;Ljava/lang/String;)V
    .locals 1

    .prologue
    .line 165
    invoke-virtual {p0}, Laa;->a()Z

    .line 167
    :try_start_0
    iget-object v0, p0, Laa;->c:Lcom/android/internal/telephony/ITelephony;

    invoke-interface {v0, p2}, Lcom/android/internal/telephony/ITelephony;->dial(Ljava/lang/String;)V
    :try_end_0
    .catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0

    .line 171
    :goto_0
    return-void
```

Fig. 4: API described in smali file.

4 EVALUATION

We carry out in-depth trials to assess the effectiveness of the suggested strategy.

4.1 The environment of simulation

Both malware and regular software samples were included in the datasets utilized in this investigation. The malware sample set among them is sourced from the well-known malware-sharing website Virus Share in the network security space. The sample set is separated into two sections based on when the data was gathered: 2791 malware samples in 2017 and 8183 malware samples collected between 2014 and

2016. There are 9058 normal software examples in the normal software sample collection, which is sourced from many official app stores including 360 Assistant, Google Play, and others. This research employs the Virus Total tool to filter the crawling regular software samples in order to assure the quality of the dataset. Consequently, 8745 is the total number of standard software samples that were employed in the experiment.

Furthermore, the collection contains almost 90% of samples that are less than 10MB. The percentage of samples greater than 20 MB is around 3%. There are two sizes of samples: one is merely 1KB and the other is 87MB. About the experimental platform, Windows 10 (64-bit) is the operating system, and every experiment in this study is performed on a PC with a dual-core 3.7GHz CPU and 8G of RAM. All of the experimental programs are created in Python and are operated via the Spyder software.

4.2 The Enhanced FP-growth Algorithm's Performance

We evaluate the performance of the enhanced FP-growth algorithm with the original FPgrowth in this work. Each sample in our dataset has a varied amount of sensitive permissions and API calls—anywhere from handful to hundreds. The quantity of often occurring item sets and the mining duration of the two methods with varying minimal support are shown in Table 1. The amount of frequent item sets mined using the enhanced FP-growth is fewer than that using the original one, as seen in Tab. 4, because of the pruning technique. Furthermore, the difference is more noticeable when the minimal support drops.

Therefore, while mining sensitive patterns, our enhanced FP-growth may effectively prevent duplication. Better yet, the enhanced FP-growth is more efficient and can finish mining faster.

Tab 4: A comparison of the two FP-growth's performance

	Original FP-growth		Improved FP-growth	
minSup	num	time(s)	num	time(s)
0.7	498	0.51	417	0.49
0.6	992	0.87	827	0.69
0.5	7384	5.23	5295	4.83
0.4	93958	25.77	34403	19.25
0.3	699861	140.61	280684	94.32

4.3 Performance of the mGBDT Algorithm

The mGBDT technique is included in our suggested approach to train the detection model. We compare it to established machine learning classification methods like Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF), and XGBoost in order to assess its performance.

The assessment metrics used in our experiment are Accuracy, Precision, and Recall, as shown by the formulae (15)–(17).

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN} \quad (15)$$

$$Precision = \frac{TP}{TP+FP} \quad (16)$$

$$Recall = \frac{TP}{TP+FN} \quad (17)$$

Of these, TP and FP stand for the quantity of malware that was accurately identified as such and the quantity of regular software that was mistakenly identified as malicious, respectively. The numbers TN and FN stand for the number of malware that was mistakenly identified as legitimate software and the amount of properly identified normal software.

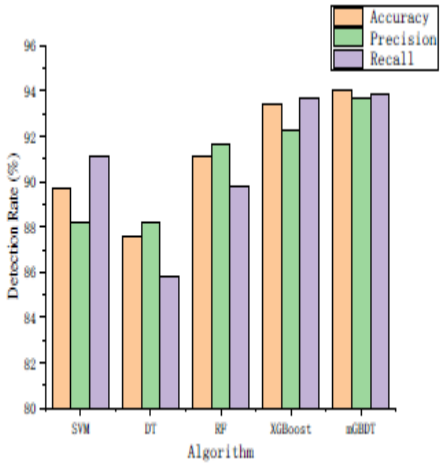

Fig. 7 The effectiveness of various algorithms.

The detection results produced by using mGBDT and comparison techniques to train the detection model are shown in Fig. 7. As Fig. 7 illustrates, mGBDT performs noticeably better than other methods. The mGBDT has shown improvements in accuracy ranging from 3%–6%, precision rate improvement from 2%–4%, and recall rate rise from 3%–8%.

In this experiment, XGBoost—an enhanced gradient boosting decision tree—demonstrates greater detection accuracy and recall. However, mGBDT is more advantageous than XGBoost in terms of accuracy. For malicious software detection systems,

this is crucial since high accuracy reduces the possibility of false positives.

4.4 The Performance of Detection

We contrast the suggested approach with other related approaches (the references [26] and [27]), which also examine permissions and API calls of Android applications, in order to verify the superiority of the suggested approach. Reference [26] examined how permissions and API updates varied between Android system versions and suggested a detailed malware detection technique for varying API levels. The use of permissions and API calls in malware and legitimate applications, respectively, was examined in Reference [27]. Based on the mapping link between permissions and APIs, it selected 50 highly sensitive API calls as differentiating characteristics.

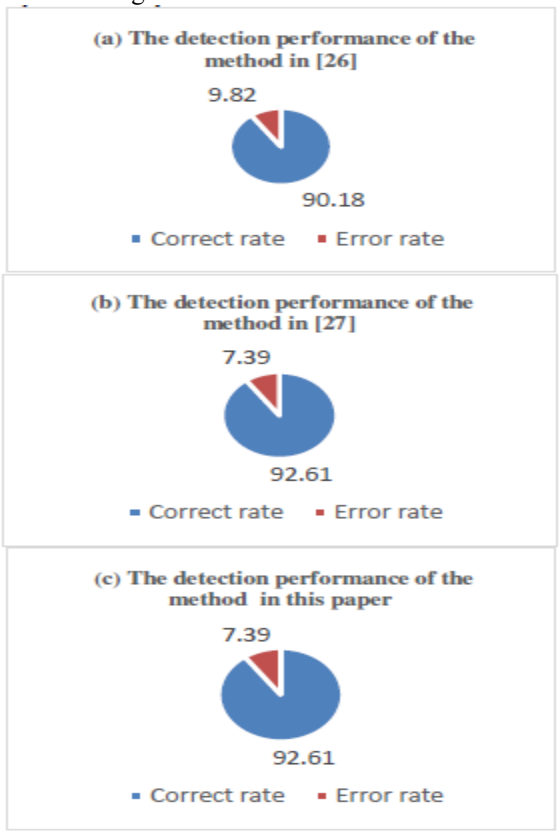

Fig. 8: Comparing the effectiveness of various approaches for detection

The detection performance of several approaches on the dataset used in this work is shown in Fig. 8. On our dataset, the compared algorithms provide accuracy values of 90.18% and 92.61%, respectively. They fall short of our suggested detection approach by 93.99%. Thus, our suggested detection approach may more accurately identify malware by examining the permission data and API call data in the APK files.

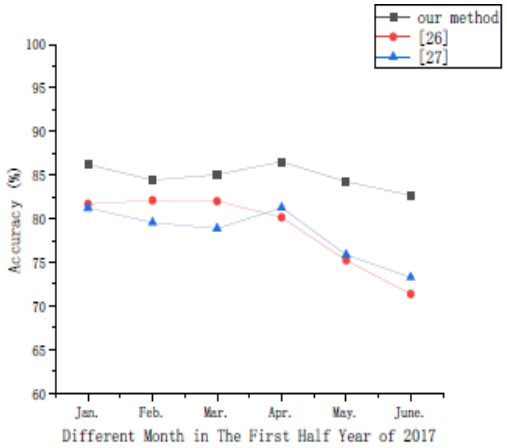

Fig.9 the effectiveness of modern virus detection.

We assess the generalization for various detection techniques on novel malware by using the samples gathered between 2014 and 2016 as the training set and the samples gathered during the first half of 2017 as the test set. The detection results for malware samples published in various months of 2017 are shown in Fig. 9. As we can see, our approach finds new viruses more effectively. It implies that when using our approach, the detecting system doesn't need to be updated regularly.

4.5 How Well API Calls and Permissions Combine

This article generates critical patterns of Android applications using both API calls and permissions. We also run studies about the use of permissions or API calls to demonstrate the efficacy of combining them.

Performance of detection models constructed with various attributes is shown in Tab. 5.

Feature Used	Accuracy (%)
Only API calls	91.25
Only permissions	88.54
Combination of permissions and API calls	93.99

A single kind of static feature has limits when it comes to differentiating between dangerous and benign programs, as Tab. 5 illustrates. We can successfully enhance the detection model's performance by merging many characteristics.

5 CONCLUSION

This research suggests a sensitive pattern-based approach for Android virus detection. We efficiently

prevent repetition by mining frequently occurring combinations of critical permissions and API requests in both malicious and benign applications using an enhanced FP-growth algorithm. To reduce feature dimensionality, we cluster the produced sensitive patterns based on text similarity and support similarity. In addition, a malware detection model with excellent accuracy and good generalization ability is constructed using the multi-layered gradient boosting decision trees approach. Nevertheless, this paper's suggested detection approach merely takes into account how often sensitive patterns appear in malicious and benign software sets. It does not take into account the variations in sensitive patterns in certain samples. In order to more precisely describe the samples, sample weight may thus be allocated to sensitive patterns in future research projects based on the quantity of API calls and other variables.

REFERENCES

[1] Sk H K. A Literature Review on Android Mobile Malware Detection using Machine Learning Techniques. International Conference on Computing Methodologies and Communication (ICCMC), 2022: 986- 991.

[2] Sharma S, Khanna K, Ahlawat P. Survey for Detection and Analysis of Android Malware (s) Through Artificial Intelligence Techniques. Cyber Security and Digital Forensics. Springer, Singapore, 2022, 321-337.

[3] Wang L, Wang H, He R, et al. MalRadar: Demystifying Android Malware in the New Era. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2022, 6(2): 1-27.

[4] Wang L, Wang H, He R, et al. MalRadar: Demystifying Android Malware in the New Era. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2022, 6(2): 1-27.

[5] Fan W, Liu D, Wu F, et al. Android Malware Detection Based on Functional Classification. IEICE Transactions on Information and Systems, 2022, 105(3): 656-666.

[6] Xin Luo, MengChu Zhou, Hareton Leun, Yunni Xia, Qingsheng Zhu, Zhuhong You, and Shuai Li. An Incremental-and-Static-Combined Scheme for Matrix-Factorization-Based Collaborative Filtering. IEEE Transactions on Automation Science and Engineering, 2016, 13(1): 333-343.

[7] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. Permission Usage to Detect Malware in Android. International Joint Conference CISIS'12-ICEUTE´12-SOCO´12 Special Sessions, 2013, 289-298.

[8] Shuang Liang, and Xiaojiang Du. Permission-combination-based scheme for Android mobile malware detection. IEEE International Conference on Communications (ICC), 2014, 2301-2306.

[9] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. Security and Privacy in Communication Networks, 2013, 86-103.

[10] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. Network and Distributed System Security Symposium (NDSS), 2014, 23-26.