DAMBA: Detecting Android Malware by ORGB Analysis

Weizhe Zhang, Huanran Wang, Hui He, and Peng Liu

***Abstract*—With the rapid development of smart devices, mobile phones have permeated many aspects of our life. Unfortunately, their widespread popularization attracted endless attacks that seriously threat users. As the mobile system with the largest market share, Android has already become the hardest hit for years. To Detect Android Malware by ORGB Analysis, we present DAMBA, a novel prototype system based on C/S architecture. DAMBA extracts the static and dynamic features of apps. For further analyses, we propose TANMAD algorithm, a two-step Android malware detection algorithm, which reduces the range of possible malware families, and then utilizes sub-graph isomorphism matching for malware detection. The key novelty of our work is the modeling of object reference information by constructing directed graphs, which is called ORGB. To achieve better efficiency and accuracy, we present several optimization strategies for hybrid analysis. DAMBA is evaluated on a large real-world dataset of $2,239$ malicious and $1,000$ popular benign apps. The detection accuracy reaches 100% in most cases, and the average detection time is less than 5s. Experimental results show that DAMBA outperforms the well-known detector, McAfee, which is based on signature recognition. In addition, DAMBA is demonstrated to resist the known malware attacks and their variants efficiently, as well as malware that uses obfuscation techniques.**

***Index Terms*—Android malware detection, object reference graph birthmarks, dynamic analysis, static analysis, sub-graph isomorphism.**

I. INTRODUCTION

ANDROID is the most widely used smartphone operating system. According to a recent report in the IDC (Internet Data Center) Quarterly Mobile Phone Tracker [34], by the first quarter of 2017, the Android market share reached 86.1%, which is much higher than those of other mobile operating systems. However, the popularization of Android smartphones is highly impaired by the prevalence of malware.

Android is a Linux-based mobile operating system that inherits the security features of Linux [44]. For instance, instead of running multiple applications with the same user, the Android system assigns a separate process and a unique ID to each app [13]. The permissions required by apps are registered in manifest files [14]. This mechanism implements the safeguard of permissions for system resources. However, hackers can still easily perform reverse engineering and invoke the underlying system library via Java Native Interface [43]. This leads to the abuse of permissions and third-party library. In addition, various Android app stores are open in nature and lack strict security review. Consequently, explosive growth of malware is flooding the market, introducing considerable threat and harm to users [7]. In view of the current malware proliferation, precautionary measures are essential. Researchers have classified Android malware into various families based on the malicious behavior [3], such as FakeInstaller and *DroidKungFu*. Four types of malware behavior detection methods exist [37].

The first type of malware detection method is inspired by static analysis. Reverse engineering is the most commonly used method in static analysis. Therefore, static methods [23], [35], [42] achieve high code coverage with low consumption. However, this type of methods lacks an actual execution path and associated execution context.

The second branch of detection method detects Android malware at run-time. Rastogi et al. [31] use different detection and automatic analysis techniques such as dynamic taint tracking and Application Programming Interface (API) monitoring to identify apps. This kind of method extracts dynamic features during the execution process of the app, and thus, it can effectively deal with code encryption and obfuscation attacks [45]. However, dynamic analysis typically lacks code coverage and presents high performance cost. The malicious code may not be triggered and identified due to insufficient inputs.

The third type of detection method is hybrid detection which combines dynamic and static analyses. Spreitzenbarth et al. [33] utilize the static method to extract the permissions required by an app and to convert the Dalvik bytecode into the smali code in order to search for dangerous features. This type of detection exhibits better accuracy and performance than the first two types.

The fourth type is machine learning-based detection, which recently exhibited the most potential. Several works have employed learning algorithms. Most notably, MaMaDroid [29] extracts API invocation to establish Markov chains of behavioral models, which in turn are used to extract features for machine learning. Drebin [3] proposes a machine learning method based on static analysis, and it is directly used on Android devices. Machine learning-based technologies generally achieve high detection accuracy, but their performance and efficiency are not considered. The training effect is overly dependent on dataset quality.

Although considerable research has been conducted on

W.-Z Zhang is with the School of Computer Science and Technology, Harbin Institute of Technology, China and with the Cyberspace Security Research Center, Pengcheng Laboratory, Shenzhen, China. (e-mail: wzzhang@hit.edu.cn).

H.-R Wang is with the School of Computer Science and Technology, Harbin Institute of Technology, China. (e-mail: wanghuanran@stu.hit.edu.cn).

H. He is with the School of Computer Science and Technology, Harbin Institute of Technology, China. (e-mail: hehui@hit.edu.cn).

P. Liu is with the College of Information Sciences and Technology, Pennsylvania State University, State College, PA, USA. (e-mail: pliu@ist.psu.edu).

TABLE I
COMPARISON OF LITERATURE WORK

Family	Resource Occupancy	Need Execution	Available for Unknown Malware Family	Resilience to Obfuscation	Accuracy	Efficiency
Static Detection	Low	No	No	No	Medium	High
Dynamic Detection	High	Yes	Yes	Yes	High	Medium
Machine Learning Based Detection	High	Not Always	Yes	Yes	High	Low
DAMBA (Hybrid Detection)	Medium	Yes	Yes	Yes	High	High

[1] Different implementations may have different conditions. We show the most common one.

malware detection, existing techniques have not been perfected in all aspects. Table I presents a comparison of the existing works. The analysis of the existing works demonstrates the following challenges:

- **Long Detection Time:** Malware detection should be completed within the prescribed time; otherwise, the user cannot be effectively warned to take countermeasures. However, the detection time of certain existing detection methods fail to meet this requirement. For example, static analysis performed by Enck et al. [16] on 1100 popular free Android apps took nearly 20 days, which is obviously impractical.
- **Inability to Deal with Code Obfuscation and Variants Attacks:** Static systems are used to analyze the resource obtained by reverse engineering. In recent years, hackers have used novel methods, such as encryption, code obfuscation, and dynamic code loading to avoid being detected by these systems [27], [30]. For some dynamic systems, such as RiskRanker [21], the simple heuristic strategy they use can easily be eluded by malware.
- **High False-Positive Rate and False-Negative Rate:** Although dynamic methods can defend obfuscation attacks, they exhibit high false-positive rate. Crowdroid [5] capitalizes on the former approaches for dynamic analysis in the Android platform. The experiments show that the system has a high rate of false-positive results and is vulnerable to attacks.

Motivated by the above challenges, we propose DAMBA (Detecting Android Malware through ORGB Analysis), a malware detection system with a C/S architecture. We found that Object Reference Graph Birthmarks (ORGB) could be used to build a detection system which is not only accurate and resilient to obfuscation, but also very efficient. Although extracting and using ORGB in a traditional way would result in low efficiency, it could be extracted and used in out novel way to achieve high accuracy and high efficiency simultaneously. In our work, before we extract and check ORGB, we first do efficient static analysis to filter out most of the impossible families. We analyze and utilize the main system activities monitored by malware, which can effectively trigger the malicious code. Then, we propose TANMAD algorithm to check sub-graph isomorphism, our new mechanisms include several optimizing strategies aim at Android malware detecting.

Experiments show that the system can effectively resist code obfuscation attacks and the variation of known malware. And the number of algorithm matching operations is effectively reduced by the static analysis, which greatly improved the efficiency and accuracy of the system. Especially, we outline the contributions of our work below:

- We analyze memory objects on the heap of Android devices to acquire object reference graphs (ORG) and extract the feature classes from different families of malware to construct ORGB. To the best of our knowledge, it is the first ORGB-based malware detecting solution applies to Android.
- We analyze the existing graph isomorphism matching algorithm and propose a novel graph-isomorphism-based algorithm, TANMAD, to detect malicious features aimed at Android platform architecture.
- We propose DAMBA, a prototype system applying static and dynamic analyses, based on C/S architecture. And we conduct a series of evaluations to verify the classification accuracy, identification accuracy, and detection efficiency of the proposed system. The experimental results show that DAMBA is highly extensible and achieves good results in detection.

The paper is organized as follows. The system architecture and the design of each part of the system are detailed in Section II. Our core algorithm is described in Section III. Our experiments are presented in Section IV, followed by a discussion of the related work in Section V. Finally, the conclusion and our future work are provided in Section VI.

II. DESIGN AND IMPLEMENTATION OF DAMBA

In this section, we present the implementation of DAMBA. This system has a C/S architecture, which consists of two parts, namely, the Android client (installed on Android devices) that extracts the object reference and the PC server that establishes the birthmark base and detects whether a suspicious app is benign. The main computational task is completed by the server to minimize impact on mobile performance. DAMBA is divided into four modules: ORG Extraction Module, ORGB Extraction Module, Family Filter Module and Subgraph Identification Module. The ORG extraction module is implemented in the Android client, whereas the remaining modules belong to the PC server.

Before detection, we establish the birthmark base according to the process illustrated in Fig. 1. First, we run malicious apps belonging to different families in Android devices and extract the ORG files using the ORG Extraction Module. Second, the ORGB Extraction Module analyzes the ORG files and outputs an ORGB file for each family. The ORGB files of the different families constitute the dynamic feature base. Meanwhile, we extract the package information for every app

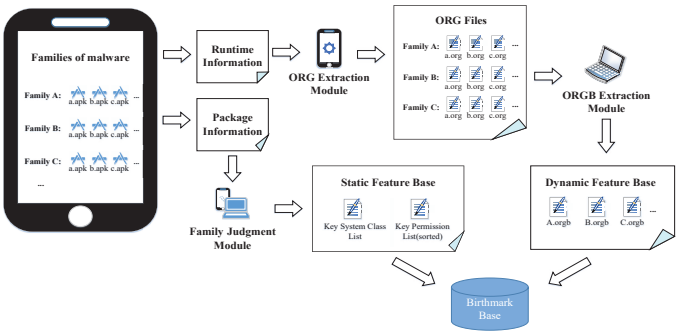


Fig. 1. Birthmark Base Establishment Flow Chart

and utilize the Family Filter Module to obtain the key system class and permission list of each family in order to develop the static feature base.

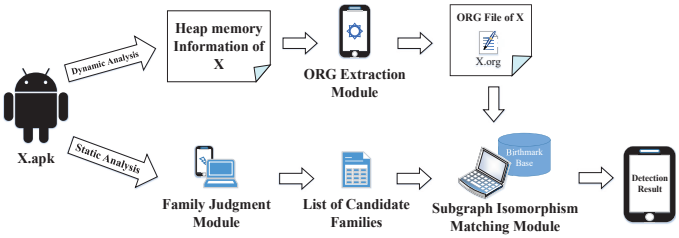


Fig. 2. Malware Detection Flow Chart

The detection process for suspicious app X is shown in Fig. 2. The memory objects of X in the heap are accessed and analyzed using the ORG Extraction Module. Simultaneously, the Family Filter Module judges the families to which X may belong and generates a list of candidate families. Finally, the list and the generated ORG file is uploaded to the Subgraph Identification Module. The identification module uses the birthmark base to search for the malicious feature in X, and the final result is sent back to the Android client.

A. ORG Extraction Module

In the ORG Extraction Module, we extract the heap file (in hprof format) of the apps to be detected. After analysis, we obtain the ORG, which is defined in Section III. It is used as the dynamic feature for detection. For the same app, the ORGs extracted at different phases vary because the malicious code is triggered by different operations. Therefore, the extraction occasion should be considered to ensure that the malicious code is executed. On the basis of the analysis of the malicious code, we discover that the malicious code causes specific system activities. Accordingly, when we train apps of a given malware family, in order to ensure that malicious code is already loaded in heap, we develop a semi-automatic script for simulation operations based on family characteristics and monitor the specific system activities in the background to extract ORG in time. For apps with special input, manual operation is adopted. Table II shows the specific system activities caused by the most classic 20 malicious families, which are monitored by us during execution. As a result, the ORG extracted can describe the malicious feature effectively. The flow of this module is shown in Fig. 3, which includes:

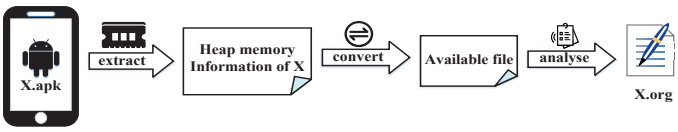
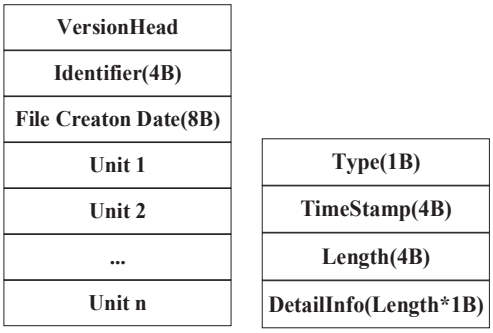


Fig. 3. ORG Extraction Module Flow Chart

Extraction of the raw heap file. Because of the differences among apps, the execution time depends on the specific functions. In general, to guarantee the fully execution of malicious code, each app runs for an average of 1 minute during training. When the specific system activities are detected, the ORG extraction module is activated. The process identifier (PID) of the specified process is first obtained, and then the built-in heap monitoring tool SDK *dumpheap* is used to export the heap file. In this step, the obtained binary heap file cannot be analyzed directly.

Format Conversion of the Heap File. The format of the raw heap file we obtained is shown in Fig. 4(a). The file starts with a version string, followed by 4-byte ID information and 8-byte creation date. After which comes the main body, which consists of units (Fig. 4(b)). Each unit stores different Java object information codes, which are sequentially stored as follows: 1-byte type-field, 4-byte time stamp, 4-byte data length n, and n-byte detail information.



(a) Raw heap file format (b) Data Unit format

Fig. 4. Format of raw heap file and data unit

Nine types are irrelevant in the detail information type field of each unit, as shown in Table III. In the conversion process, the irrelevant types are initially identified and then deleted from the raw heap file. Subsequently, the conversing process outputs the new heap file for the next step.

Analysis of Available Heap File. Malicious code is generally embedded in normal APK files. When the code runs in Android phones, the objects and the reference relationship between these objects are created in the heap memory. Useful data, including the created objects, the reference relationship between these objects, and the times of reference, are extracted as the feature of the app. We implement MHAT, a heap file analysis tool for Android. MHAT is based on the principle of *JHAT*, a built-in tool of JDK for PC. The extraction process involves binary heap file parsing, class reference analysis, and ORG file creation.

TABLE II
THE KEY OCCASIONS OF 20 MALWARE FAMILIES

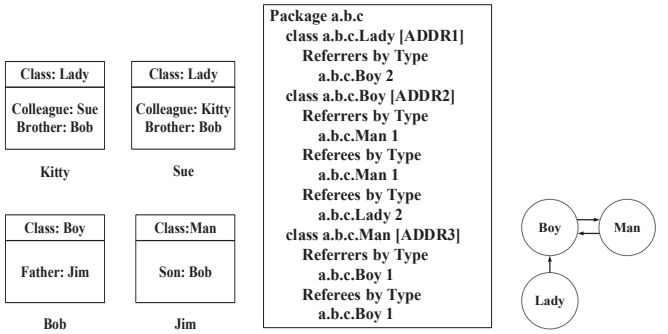
Family	Monitor Activities	Malicious Behaviors
ADRD	INTERNET Request, WALLPAPER_SETTINGS_ACTION	It encrypts the stolen information and sends it through a local proxy.
AnserverBot	Upgrade dialog, ALTERNATIVE_CATEGORY	It displays a fake upgrade dialog to lure the user to install a hidden payload.
BaseBridge	SMS_SEND, UPDATED, SEARCHABLES_CHANGED	It attempts to send premium-rate SMS messages to predetermined numbers.
Bgserv	INTERNET Request	It opens a back door and transmits information from the device to a remote location.
DroidDream	INTERNET Request	It downloads additional malicious programs without the user's knowledge.
DroidDreamLight	SETTINGS_ACTION	It steals mobile-specific information, such as IMEI number that it then uses for malicious activities.
DroidKungFu1	INTERNET Request, DEFAULT_ACTION	It installs a back-door component onto a device, and forwards data stolen from the device to a remote server.
DroidKungFu2	INTERNET Request, DEFAULT_ACTION	It sends sensitive information to an attacker and includes backdoor functionality. It also exploits vulnerabilities to gain root access.
DroidKungFu3	SETTINGS_ACTION, INTERNET Request	It forwards confidential details to a remote server.
DroidKungFu4	SETTINGS_ACTION	It collects information from the device it was installed on.
YZHC	INTERNET Request	It performs a number of actions of a malicious hacker's choice on your PC.
Zhash	LOGIN_ACTION, DEFAULT_ACTION	It drops a binary file that uses exploit code in an attempt to root the device.
Zsone	SMS_SEND	It sends SMS messages to premium-rate numbers related to subscription for SMS-based services.
Geinimi	SETTINGS_ACTION INTERNET Request	It variants harvest details of the affected device and forwards them to a remote location.
GoldDream	INTERNET Request	It steals information from Android devices.
Gone60	INTERNET Request, SYNC_ACTION	It uploads and sends to a remote server contacts, messages, recent calls, browser history stored on the phone.
jSMSHider	ADD_SHORTCUT_ACTION, SMS_SEND	It opens a back door on Android devices.
Kmin	SMS_RECEIVED	It variants display a message as a decoy, while silently performing multiple malicious routines.
Pjapps	INTERNET Request	It opens a back door on the compromised device. It retrieves commands from a remote command and control server.
Plankton	INTERNET Request	It steals system information. In addition, they download an additional file onto the device.

TABLE III
IRRELEVANT TYPES IN UNIT

Type Name	Hexadecimal Mark
HPROF_HEAP_DUMP_INFO	0xfe
HPROF_ROOT_INTERNED_STRING	0x89
HPROF_ROOT_FINALIZING	0x8a
HPROF_ROOT_DEBUGGER	0x8b
HPROF_ROOT_REFERENCE_CLEANUP	0x8c
HPROF_ROOT_VM_INTERNAL	0x8d
HPROF_ROOT_JNI_MONITOR	0x8e
HPROF_UNREACHABLE	0x90
HPROF_PRIMITIVE_ARRAY_NODATA_DUMP	0xc3

The ORG is extracted in the Android client because the size of the exported raw heap file is extremely large. For example, a lightweight build-in Android calendar generates a heap file with a size of approximately 8 MB. Furthermore, at least 10 processes are running simultaneously in a normal phone. If each process generates a file of at least 8MB and sends it to the PC server, a huge burden is placed on the Internet service provider of the Android client. Therefore, only the useful information is extracted to establish the ORG.

The three classes and the reference relationships in Fig. 5(a) are taken as examples, *Kitty* and *Sue* are the objects of the *Lady* class, *Bob* is the object of the *Boy* class, and *Jim* is the object of the *Man* class. After extraction, MHAT outputs the ORG file of this case, as shown in Fig. 5(b). The first line indicates the package of the three classes, and the second line is the class name followed by the heap address. In each class, types of reference relationship exist, namely, referee and referrer, as well as reference times. Referee relationship refers to the class the relationship references, whereas the referrer relationship refers to the class referencing the relationship. To construct



(a) A Simple Example (b) Format of ORG (c) Digraph ORG

Fig. 5. Format of raw heap file and data unit

the digraph, the classes are used as nodes, and the reference relationships as directed edges. The ORG constructed is shown in Fig. 5(c). Notably, although a reference relationship exists between Kitty and Sue, the edge between them is ignored because it represents a self-referencing relationship.

B. ORGB Extraction Module

We utilize the ORGB (defined in Section III) as the feature of the malware families, and the largest common part of the ORGs that belong to the same malware family. The ORGB consists of two parts: the structurally matched part and the isolated points obtained by name matching. Given that two ORG belong to the same family, the ORGB of the family is extracted by, initially using name matching for the isolated points, which cannot be identified by structural matching as they are not connected to an edge. Subsequently, the isolated points with the same name are added to the ORGB as a common isolated point, and the remaining parts of ORGs are matched structurally. Finally, the largest matched sub-graph is added to the ORGB. The obtained ORGB of the family is shown in Fig. 6. The same process is repeated for each malware family to establish the dynamic feature base.

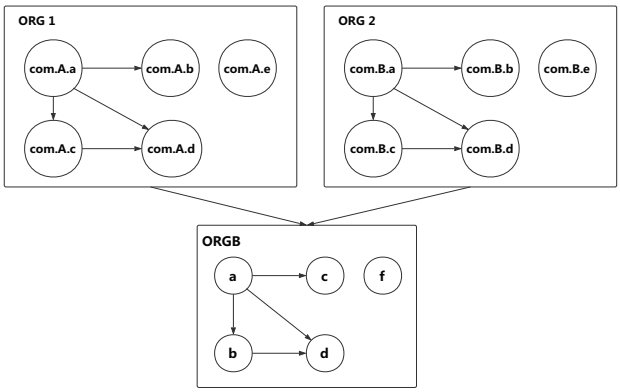


Fig. 6. ORGB Extraction Diagram

C. Family Filter Module

Given a malicious app X belonging to family Y, we find that the matching between X and Y is rapid and it's slow between X and other families. The reason is that the recursion mechanism in our algorithm needs to traverse every state until the termination condition is satisfied (it will be discussed in Section III). To reduce the range of families to which X may belong, we design the Family Filter Module to establish the static feature base and filter out the families to which the test app is impossible to belong statically before graph matching. After filtering, the candidate families are obtained, and then, we calculate the likelihood of them to improve the accuracy of the judgment.

Different malware families contain different permissions and system classes, whereas apps in the same family have similarities. Therefore, the key permissions and classes of each malicious family are acquired and then used to analyze unknown apps. Key permissions refer to permissions required by all apps in a certain family. Key system classes are the system classes invoked by all apps in a certain family at runtime.

As mentioned in Section I, the app should declare the required permissions in the app manifest file. For example, when an app needs to access the network, it needs to add *<uses-permission android: name="android.permission.INTERNET" />* in the manifest file [22]. In the filtering process, a family is added to the candidate list when the test app contains all the key permissions of the family. As shown in Fig. 7, the permissions from the *AndroidManifest.xml* files of each family are first extracted. After obtaining the common permissions, we remove the useless ones to obtain the key permissions list. Finally, the contribution degree and the distinction degree of each key permission (discussed in Section III-B) are calculated and the key permission feature file is created. Similar to the filtering of permissions, we derive the key system class of each family. We establish the static feature base by collecting the key system class and key permission lists of each family.

In the static analysis, the permission list of X is obtained and compared with that of every family in the static feature base. If the list does not include one or more terms of a family, then X does not belong to the family. After filtering, the candidate families are sorted according to the likelihood PX (which is discussed in Section III-B). The entire flow is shown in Fig. 8.

D. Sub-graph Identification Module

In this module, we match the ORG of app X with each candidate family's ORGB obtained by the ORGB Extraction Module. In view of the existence of isolated points, name matching should be used for pre-processing. If the number of successfully matched nodes reaches the set threshold, the process returns a success message, then X belongs to the matched family. For the dynamic analysis, we propose the TANMAD algorithm, which is discussed in Section III.

III. TANMAD Algorithm

Inspired by traditional host-oriented methods [28], we utilize the object reference graph and the object reference graph birthmark [6], [12] to mark the dynamic features of apps and malware. We give the definition of ORG and ORGB below:

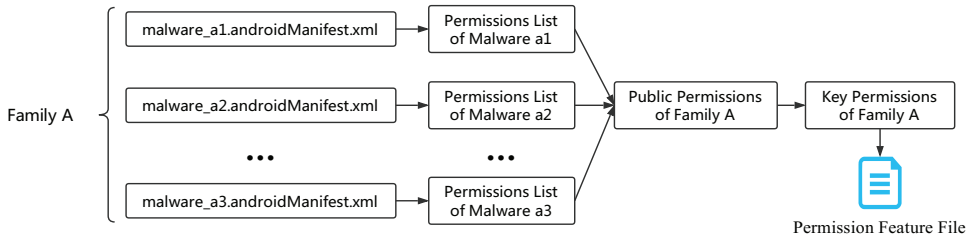


Fig. 7. Obtaining of Key Permissions

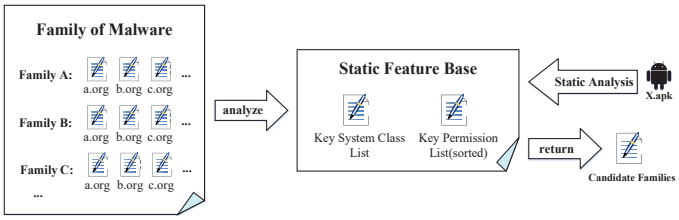


Fig. 8. Family Filter Module

Definition 1: (ORG) Given a set of vertices N, which denotes classes, and a set of edges $E \in (N \times N)$, which represents the reference relationship between objects, ORG is a 2-tuple (N, E).

Definition 2: (ORGB) Given programs m, n, and input s, when s is taken as input to m and n, the process separately extracts ORG_m and ORG_n, $ORGB_m$ is the sub-graph of ORG_m, and $ORGB_m$ corresponds to the dynamic birthmark iff s.t.

- If n is part of m, then $ORGB_m$ and ORG_n are sub-graph isomorphic;
- If n is not part of m, then $ORGB_m$ and ORG_n are not sub-graph isomorphic.

In order to avoid the limitations of dynamic analysis, we adopted TANMAD, a two-step Android malware detection algorithm that includes two steps. First, the candidate families of the test app are inferred by static analysis. Second, sub-graph isomorphism detection is performed between the ORGs and ORGBs.

A. Review of Isomorphism Patterns

The patterns of isomorphic graphs include isomorphism, sub-graph isomorphism, and monomorphism. Isomorphism is the strictest pattern, in which the vertex mapping of two graphs should bebijective, and the relevant vertices should demonstrate edge retention properties. In sub-graph isomorphism, a sub-graph of a graph is isomorphic with another graph. Monomorphism has a weak constraint as it allows different vertices of the first graph to be mapped to the same vertex on the second graph.

We use sub-graph isomorphism for ORG matching because in real scenarios, noisy data inevitably exists in graph construction and the use of Isomorphism increases omissive judgment rate. Furthermore, the weak constraints of Monomorphism can increase misdiagnoses rate. Isomorphism requires that the two given graphs are not only structurally isomorphic but also consistent with the attributes of vertices and edges.

B. Judgment of the Candidate Families of Suspicious App

The first step of the TANMAD algorithm is pre-processing. As mentioned in Section II-C, the candidate families of detected apps are judged to avoid the practicability loss caused by the long detection time. After the candidate list is acquired by the Family Filter Module, permission feature file is created, as described in this section.

The permission feature file contains the key permissions of the given family. In addition, each permission includes two attributes: **contribution degree** and **distinction degree**. The format of the permission feature file is shown in Fig. 9.

<Permission 1 , Contribution degree f1 , Distinction degree e1>
<Permission 2 , Contribution degree f2 , Distinction degree e2>
...
<Permission N , Contribution degree fn , Distinction degree en>

Fig. 9. Format of Permission Feature File

Given permission *i* and malware family set *C*, the contribution degree f_i^k $(k \in C)$ is expressed in equation 1, which is the positive correlation with a frequency of *i* in family *k*. In equation 1 *m* denotes the number of apps which include permission *i* in family k, and *n* denotes the total number of apps in k.

$$f_i^k = \frac{m}{n} \tag{1}$$

The distinction degree of *i* is expressed in equation 2, which denotes the rarity of the permission. In equation 2, *t* denotes the number of families, and *s* denotes the number of families that contain permission *i*.

$$e_i = \frac{t}{s} \tag{2}$$

For example, if family k contains 10 apps in the static feature base, and the permission *INTERNET* occurs in 8 of them, then $f_{INTERNET}^k = 0.8$. If 10 families are obtained in the base, and 5 of which contain *INTERNET*, then $e_j = 2.0$.

When classifying suspicious apps *X*, we first obtain the permission set and successively match it with the families in the static feature base. In addition, given the key permission set l_k of k, the suspicious app *X*, and the permission set U of X, the judgment consists of two phases. First, each family k in *C* is checked whether $l_k \subset U$. If so, add k in the candidate

set C_Y. Second, the likelihood that *x* belongs to *k* is calculated using equation 3.

$$P_x^k = \Sigma_{i\in(I \bigcap U)}(f_i^k \times e_i^k), k \in C_Y \tag{3}$$

Through this process, the P_x of every family is acquired, and the results are sorted in descending order. The top families are the families to which *X* most likely belong.

C. Sub-graph Isomorphism Matching

The second step of TANMAD is the sub-graph isomorphism matching between the ORG and the ORGB. In this matching process, we utilize a parameter, λ, to deal with different matching tasks. In this addition, on the basis of the fundamental strategy of the VF2 algorithm, we adopt State Space Representation (SSR) [11] to describe the matching process and propose two optimization strategies based on the actual situation of Android malware detection.

1) Matching Precision Parameter: As discussed in Section III-A, we use sub-graph isomorphism for ORG matching. However,successful matching is difficult to realize in practice, primarily because the time for malicious code running in the memory is insufficient, resulting in the incomplete creation of references between objects. Therefore, we relax the matching constraints by proposing parameter λ whose values are in the range $(0, 1]$. This parameter denotes the ratio of the matched vertexes in the total vertexes of ORGB.

Although the numbers of classes contained in the different families of the ORGB has a large difference, λ should to be set appropriately according to the specific situation. If the ORGB is large, a match score of 80% indicates that the given app is much likely to belong to the detected family. Therefore, we utilize a lower λ to reduce the false-negative rate. If the ORGB is small, we use a large λ to reduce the false-positive rate. In Section IV, we discuss the appropriate value of λ acquired by experiment.

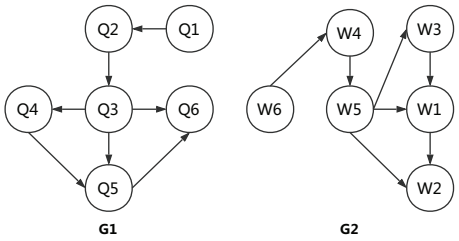


Fig. 10. Digraph G_1 and G_2

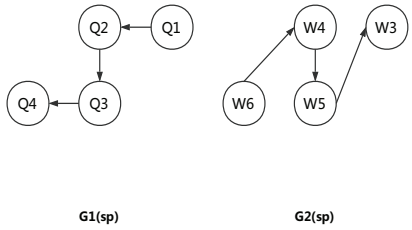
2) SSR and Two Pruning Strategies: Two detected digraphs, $G_1(N, E_1)$ and $G_2(M, E_2)$ as shown in Fig. 10 are taken as examples to provide basis for the following discussion. The isomorphic mapping solution K is described as a set of vertex pair, such as (q, w), which denotes the mapping of vertex q in G_1 and vertex w in G_2. Using for reference of VF2 algorithm, we transformed the sub-graph isomorphism matching problem in malware detection into the search for the optimal solution K.

The search for K is described effectively by State Space Representation [10], [32]. Each state s in the matching process corresponds to a partial mapping $K(s)$ $(K(s) \subseteq K)$. In the matching process, a search tree, in which each node denotes a state, is constructed. If state s can be converted into state $s^{'}$ by adding an available vertex pair, then s is the parent node of $s^{'}$ in the tree. The available vertex pair set is expressed as the candidate mapping set, $H(s)$. We use $G_1(s)$ and $G_2(s)$ to denote the projection of $K(s)$ onto G_1 and G_2, respectively; $V_1(s)$ and $V_2(s)$ are the vertices of G_1 and G_2, respectively. $E_1(s)$ and $E_2(s)$ are the edge sets of G_1 and G_2, respectively.

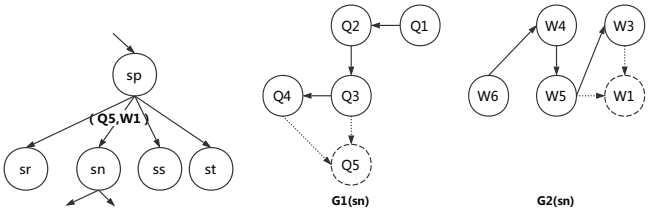
For the convenience of statement, we assume that an intermediate state sp exists. The matched sub-graph $G_1(sp)$ and $G_2(s)$ are shown in Fig. 11(a), and the partial mapping is presented as follows.

$$K(sp) = \{(Q_1, W_6), (Q_2, W_4), (Q_3, W_5), (Q_4, W_3)\}$$
$$V_1(sp) = \{Q_1, Q_2, Q_3, Q_4\}$$
$$V_2(sp) = \{W_6, W_4, W_5, W_3\}$$

Evidently, state sp has four feasible states in $H(sp)$. These states correspond to the four child nodes in the search tree shown in Fig. 11(b). With node sn taken as an example, after the vertex pair (Q_5, W_1) is adding, the process traverses to sn, and the matched sub-graphs are $G_1(sn)$ and $G_2(sn)$, as shown in Fig. 11(c). Instead of searching for all the feasible states, we propose two optimization strategies based on Android conditions for pruning to reduce the scale of $H(s)$.



(a) $G_1(sp)$ and $G_2(sp)$



(b) Local Search Tree of State sp

(c) $G_1(sn)$ and $G_2(sn)$

Fig. 11. A Middle State sp of Searching

The first strategy is based on semantic compatibility. The VF2 algorithm uses five feasible rules to restrict the syntax of the graphs. However, it gradually reveals a limitation with a large-scale matching task. Therefore, we introduce semantic compatibility to improve detection efficiency. In accordance with Android feature, no two nodes will have the same name in an ORG, and the system class name is unchangeable in code obfuscation. Therefore, we add a semantic rule for pruning $H(s)$: if the system class nodes of the two graphs have the same name, then they will be matched directly. Using the

invariability of the system class name is of considerable value. Fig. 12 shows the improvement of semantic pruning.

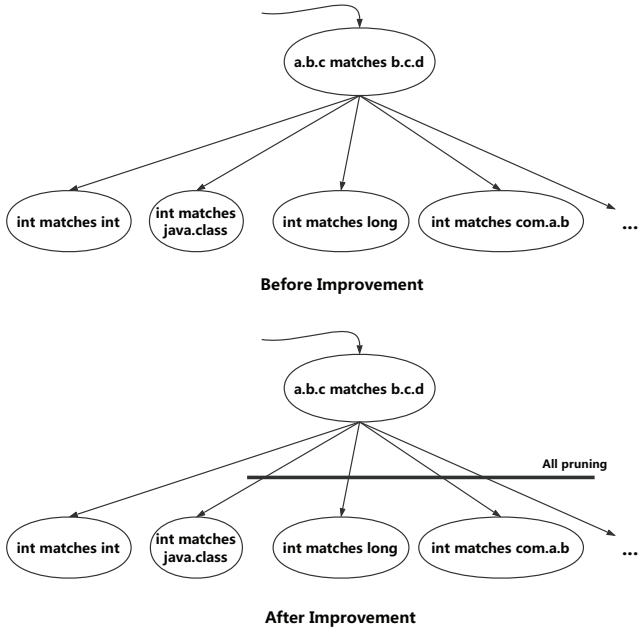


Fig. 12. Improvement of Semantic Pruning

The second strategy is sorting $H(s)$. In the malicious part of the ORG, the vertexes with a large degree are much likely the core part, which is seldom associated with the benign parts. Therefore, the optimal vertex pair is degree similarity. To take advantage of this, we sort the mapping sets $H(s)$ by the order of degree similarity. In this manner, the matching process searches the tree in depth-first order recursively until the final result is obtained:

$$K = \{(Q_1, W_6), (Q_2, W_4), (Q_3, W_5), (Q_4, W_3), (Q_5, W_1), (Q_6, W_2)\}.$$

3) Complexity Analysis: The pseudo code of the TANMAD algorithm is shown in Algorithm 1. As stated above, our algorithm consists of two steps. First, the birthmark base M is established, and the candidate families of the suspicious app are discriminated. Each detected candidate family invokes the function *IMPROVEDMATCH*. In this function, K is initialized by the system classes occurring in the two graphs. The matched vertex pair is calculated to obtain the candidate mapping set $H(s)$ of the current state. Next, after pruning the sorted $H(s)$ with the optimization strategies and rules, the search tree is traversed in depth-first order, and $H(s)$ is calculated recursively.

The time and space cost of the first part can be ignored because they only depend on the constant level of the birthmark base resources. Therefore, we only discuss the costs of the matching process in this section. The time cost consists two parts: the time for traversing the search tree and the time for calculating each state.

The best condition for traversing is that in which the candidate mapping set for each state only contains one pair. In this case, the number of the nodes to be calculated is the ORGB vertex number, which is denoted by N. The worst condition is that in which all the nodes in the $N-$depth search tree are

Algorithm 1 TANMAD Algorithm

Require: suspicious app X, birthmark base M, state s, initial state s_0, precision control parameter λ
Ensure: the isomorphic mapping solution K

```
function JUDGE(X, M)
    foreach m in M do
        if X does not belong to m then
            continue
        else
            IMPROVEDMATCH(s_0)
        end if
    end for
end function

function IMPROVEDMATCH(s)
    K is initialized by the system classes occurring in the two graphs only when it is the first run here
    if |K(s)| ≥ λ|G_2| then
        return Success Match
    else
        Find H(s) which is the set of possible pairs for s
        if H(s) is ∅ then
            return False
        end if
        Sort(H(s)) based on degree of the node in descending order
        foreach h in H(s) do
            if all rules are satisfied for h added to s then
                s' ← put h into K
                IMPROVEDMATCH(s')
            end if
        end for
        Restore data
    end if
end function
```

calculated. In this case, the total number of nodes in the tree is $1+N!\sum_{d=1}^{N-1}\frac{1}{d!}$, which is $O(N!)$ because $\sum_{d=1}^{N-1}\frac{1}{d!}$ is less than 2. After pruning, the cost of sorting can be ignored because the size of $H(s)$ is constant. Therefore, the time cost T for each state includes three parts: computation time for $H(s)$ (T_P), rule matching time T_R, and processing time for a new state (T_{NEW}). Obviously, $T = T_P + T_R + T_{NEW} = O(N)$.

Afterward, we derive the traversal and computational costs, and the total time complexity of the TANMAD algorithm is their product. The best case is: $O(N) * O(N) = O(N^2)$, whereas the worst case is: $O(N!) * O(N) = O\big(N \times (N!)\big)$.

In the algorithm, we adopt the sharing data structure. The storage space required by each state is constant. The depth of the search tree traversed in the depth-first order is no more than N. Therefore, the space complexity is $O(N)$.

IV. EVALUATION

We conduct five sets of extensive experiments using real-life data to evaluate the performance of our work adequately. In the first set of experiments, the metrics used for evaluating the

static analysis are filtering percentage and judging accuracy. In the second set of experiments, the metrics used for assessing the impact of λ are the false-negative and false-positive rate. In the third set of experiments, we test the ability of the proposed detection system to resist code obfuscation attack and variant attack. In the forth set of experiments, we systematically evaluate the detection efficiency of DAMBA by the average detecting time. In the last set of experiments, we compare DAMBA with one of well-known Android malware detection engines, called Mcafee [25], which is based on signature recognition. This is a signature-based product that is still being updated. To evaluate the effectiveness of proposed method, we use the confusion matrix to measure accuracy, recall rate, false positive, false negative, precision rate and F-measure in our experiments.

A. Experimental Setup and Dataset

In our experiment, we use a Nexus S mobile phone which runs on Android 4.3 (API 18) and has an ARM processor with four cores. Our PC server is an Intel i7-4790 computer with four cores running at 3.6 GHz each.

The datasets we use consist of three parts. The first part includes 1139 malware samples from Genome Project of the Android Malware [2](from August 2010 to October 2011) and Contagion [9](from 2011 to 2017), which belong to 20 malicious families. The second part includes 1,100 malicious samples from 2018 collected by Contagion. The third part includes $1,000$ popular benign apps from the Baidu App Store [4].

TABLE IV
EXPERIMENT GROUPING FOR THE SELECTED 20 FAMILIES

Family	Used for: ORGB Extraction/Testing
ADRD	5/4
AnserverBot	94/93
BaseBridge	61/61
Bgserv	5/4
DroidDream	8/8
DroidDreamLight	23/23
DroidKungFu1	13/12
DroidKungFu2	15/15
DroidKungFu3	155/154
DroidKungFu4	48/48
YZHC	5/4
Zhash	6/5
Zsone	6/6
Geinimi	35/34
GoldDream	24/23
Gone60	5/4
jSMSHider	8/8
Kmin	26/26
Pjapps	28/28
Plankton	5/4
Total	575/564

According to the classification and scale of different datasets, the samples of the first part and the third part are used for experiment 1, experiment 2, experiment 3, experiment 4. The data of the second part and the third part are used for experiment 5. Because the spread of different types of malware is different, the proportion in the dataset is inherently uneven. To reduce the impact on validity, we consider each type of virus sample separately. We randomly divide each family in the first part of datasets into two groups as shown in Table IV. The first group is used for extracting the ORGB, and the second group is utilized together with the benign apps for testing.

In the experiment, the apps are initially installed, and the ORG in the memory is obtained. The ORG is then matched with the candidate ORGB, and the final result is derived.

B. Evaluation of Detection Capability

DAMBA uses static analysis for family classification in the first step and employs the dynamic behavior features for further graph-isomorphism-based detection. It can effectively resist known malware attacks and the variants attacks, as well as malware that utilizes code obfuscation. The experimental results are presented in the following subsections.

Experiment 1 (Family judgment using static analysis): We collect $1,000$ benign and 564 malicious apps for the test group. This experiment is two-folds: We first filtering out the impossible malware families by static analysis, and then calculate the likelihood of every candidate family.

The metrics used in the first part is the percentage of the families to which the test app is impossible belong filtered out through static analysis, which is shown in Table V. In the filtering process, we compare the test apps with 20 malicious families in the static feature base and filtering out the families with different permissions or system classes from those of given test app. Therefore, the total number of comparisons equals the size of the test group times 20, and the filtering percentage equals the times families are filtered out divided by the times of comparisons.

TABLE V
THE RESULT OF FAMILY FILTERING

	Benign App	Malicious App	Total
Number of App	1,000	564	1,564
Times of Comparisons	20,000	11,280	31,280
Times of Filtering out	12,636	7,103	19,739
Filtering Percentage	63.18%	62.97%	63.10%

The experimental results show that the filtering percentages for benign and malicious apps are all approximately 63%, which is acceptable but not ideal. These less than ideal results are attributed to the host apps of malware having similar functions, so the correlation between permissions is strong. Another reason for such a result is the abuse of permission for benign apps, that is, the apps frequently require permissions without actually using it, thereby considerably influencing the result.

The metrics used in the second part is the judging accuracy of the top 5 candidate families of test apps. We calculate the likelihood of every candidate family after filtering out the impossible ones. Noisysounds.apk (a malicious app belonging to the ADRD virus family) is taken as an example and the likelihood calculation results are shown in Fig. 13. The top 5 possible categories are ADRD (167.74), DroidKungFu3

TABLE VI
DETECTION OF FALSE-POSITIVES RATE AND FALSE-NEGATIVES RATE IN DIFFERENT λ FOR DIFFERENT SCALE OF ORGB

Family	Node Number of ORGB Non-system/System Class	Edge Number of ORGB	False-Positive/False-Negative Rate(%) of Different λ			
			0.95	0.9	0.85	0.8
YZHC	30(8/22)	28	0.83 /85.71	0.83 /57.14	5.00 /0.00	5.00 /0.00
SndApps	26(4/22)	31	0.00 /33.33	3.33 /0.00	5.83 /0.00	6.67 /0.00
PjApps	8(3/5)	5	1.67 /0.00	1.67 /0.00	2.50 /0.00	2.50 /0.00

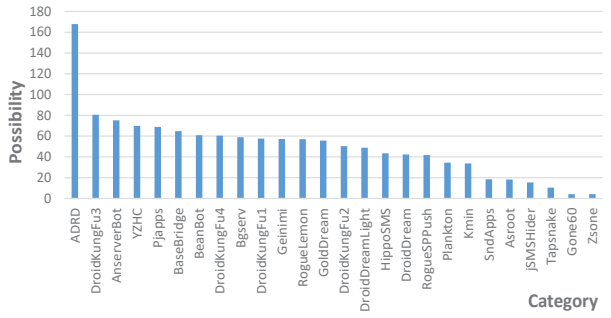


Fig. 13. Likelihood Calculation of Candidate *Families for com.noisysounds.apk*

(80.42), AnserverBot (75.21), YZHC (69.83), and Pjapps (64.88). Thus, the given app is classified correctly.

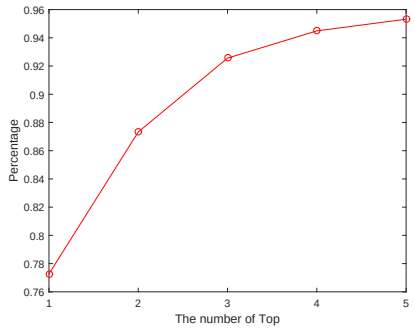


Fig. 14. Classifying Accuracy Based on the Top Likely Families

Fig. 14 shows the classification accuracy based on the top candidate families for the 564 malicious apps of 20 families. The accuracy reaches 95% when the top 5 families are considered.

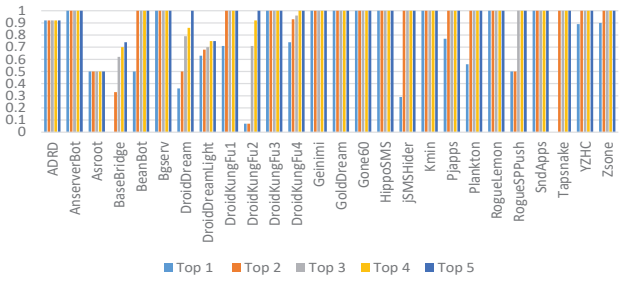


Fig. 15. Accuracy of Each Family Considering Top 1-5

Fig. 15 presents the classification accuracy for each family when the top 5 candidate families are considered. The experimental results indicate that the classification accuracy rates for most families are close to 1; only few families obtain low accuracies, such as *Asroot* and *BaseBridge*. The reasons for the low accuracy of these families are the low number of required permissions and the lack of similarities between different apps.

Experiment 2 (Impact of λ on false-negative rate and false-positive rates): Parameter λ is adopted to control matching precision. As mentioned in Section III-C1, this parameter affects detection accuracy. Therefore, it should be set appropriately according to the specific situation. Fig. 16 shows that with a decrease in λ, false-positive rate increases, and false-negative rate decreases. We take the trade-off analysis of the experimental data and adopt the condition when the value of λ is 0.87, a low false-positive and a low false-negative rates are obtained simultaneously.

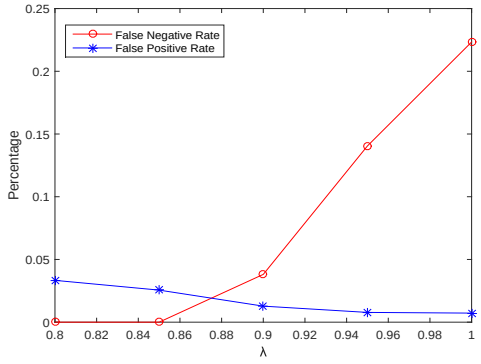


Fig. 16. Variation of the False-Negative Rate and the False-Positive Rate

The optimal value of the parameters are determined by comparing the accuracy rate of different ORGB scales in each family with different parameters. Table VI shows the comparison of three different ORGB scales, which can lead the conclusion that a low λ produces good results for a large-scale ORGB, whereas a high λ generates good results for a small-scale ORGB.

Experiment 3 (Resistance against obfuscation attacks and variant attacks): In this study, we adopt graph-isomorphism-based detection, which can efficiently re-

sist obfuscation attacks. *Plankton* is taken as an example. As shown in Table VII, the obfuscated *Plankton* malware *Plankton* can still be identified by DAMBA. For ease of expression, we use *X* to denote similar parts.

TABLE VII
THE MATCHING RESULT OF OBFUSCATED *Plankton*

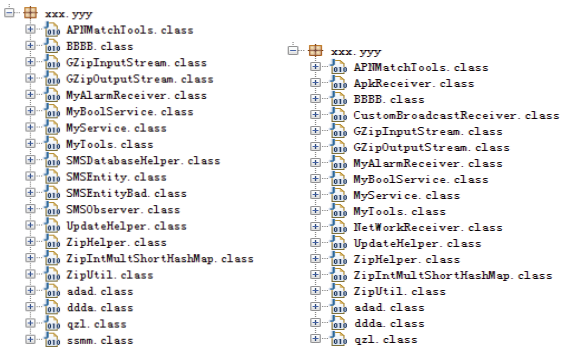
Class Type	Before Obfuscation	After Obfuscation
System	X.ContextImpl X.ContextImpl X.Application X.Class X.ActivityThread X.BinderProxy X.String X.ManagerProxy X.HashMap X.AtomicBoolean X.AsyncTask X.Collections int[]	X.ContextImpl X.ContextImpl X.Application X.Class X.ActivityThread X.BinderProxy X.String X.ManagerProxy X.HashMap X.AtomicBoolean X.AsyncTask X.Collections int[]
Non-system	X.AndroidMDKService X.BackgroundJarLoaderTask X.CustomException	X.c X.a X.e

In addition, Table VIII shows the detection results for the *Plankton* with different values of λ. DAMBA can effectively deal with obfuscation attacks.

TABLE VIII
RESULTS OF *Plankton* IN DIFFERENT λ VALUES

λ	0.95	0.9	0.85	0.8
Accuracy	9/9	9/9	9/9	9/9
False Positive Rate	0/46	0/46	0/46	0/46
Recall	9/9	9/9	9/9	9/9

The malware family *ADRD* has two variants: *ADRD.A* and *ADRD.B*. As shown in Figs. 17(a) and 17(b), these variants exhibit similarities and differences in terms of package structure. The ORGB of *ADRD.A* is matched with the ORG of *ADRD.B*. Findings indicate that the detection is successful, thereby demonstrating the capability of DAMBA.



(a) Structure of *ADRD.A* (b) Structure of *ADRD.B*

Fig. 17. Program Structure of variants of *ADRD*

Experiment 4 (Detection Efficiency): In the detection process, the candidate families of the detected apps are initially assessed to avoid the practicability loss caused by the long detection time. Subsequently, a thread for the graph matching of each candidate family is started. If one of these threads is finished, then the detection of the test app is likewise finished. In our experiment, DAMBA cost 2.2 h for detecting 1564 test apps. The detection time distribution is shown in Fig. 18.

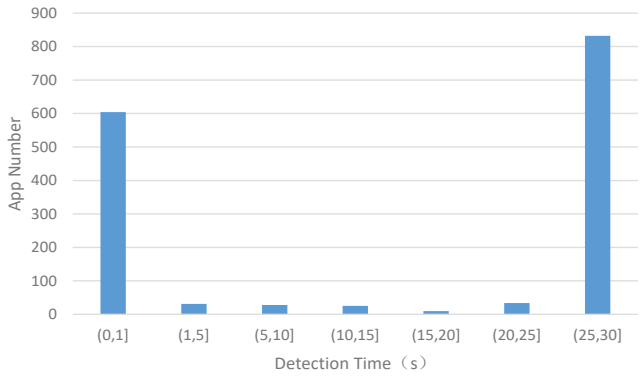


Fig. 18. Detection Time Distribution

The minimum detection time of the system is less than 1 s, whereas the maximum detection time is 30 s. Therefore, the average detection time is less than 5 s. The experimental data show that the detection time of most malware is in the interval (0,1), whereas the detection time for most benign apps is in the interval (25,30).

Experiment 5 (Effectiveness Comparison with McAfee:): Finally, we make the effectiveness comparison of Android malware detection capability between DAMBA and McAfee. In this study, we use 1,100 malware samples collected in 2018 and 1,000 benign apps crawled from the Baidu app store. We develop a script to invoke the official API of McAfee provided by VirusTotal [1] for detection. The confusion matrix and the evaluation metrics comparison of both systems are shown in Table IX and Table X respectively.

The experimental results show that DAMBA achieves better effectiveness than McAfee. From the average of accuracy, recall, precision, and F-measure, DAMBA perform over about 3%, 1%, 5%, and 3% than McAfee respectively.

TABLE IX
THE CONFUSION MATRIX OF DAMBA AND MCAFEE

(a) The confusion matrix of McAfee

Result	Benign	Malware
Benign	904	96
Malware	19	1081

(b) The confusion matrix of DAMBA

Result	Benign	Malware
Benign	951	49
Malware	16	1084

TABLE X
THE COMPARISON OF EVALUATION METRICS

Detector	Accuracy	Recall	Precision	F-measure
McAfee	0.9381	0.9755	0.9124	0.9429
DAMBA	0.9690	0.9855	0.9568	0.9709

C. Case Study: BaseBridge

A weakness of many static detecting approaches to malware detection is their inability to analyze the obfuscated code. Because once obfuscated, the original format and semantics of the code are difficult to detect. Our method addresses this shortcoming by adopting the combination of dynamic and static methods. The advantage of this is to reduce the cost of the detection process and effectively improve the accuracy rate.

As an example, a malicious app named 'soft.apk' (MD5: B90F95C3A296408DAC451B58AD8FEC6F) a sample from the BaseBridge family. (BaseBridge [36] is a Trojan horse that attempts to send premium-rate SMS messages to predetermined numbers.) Before 'soft.apk' was installed, there were no other third-party apps in the system. The system broadcast information at this time is shown in Fig 19(a). When it is installed, it automatically downloads and installs an unknown update package at startup, then calls the system SMS service to send a message to a premium number. The system broadcast shown in Fig 19(b) is issued when malicious code is triggered, which completely reveals the malicious behavior of 'soft.apk'.

```
root@android:/ # pm list packages -3
root@android:/ # dumpsys |grep BroadcastRecord
  BroadcastRecord{537df604 com.android.internal.telephony.gprs-data-stall}
  BroadcastRecord{53789234 android.intent.action.TIME_TICK}
  BroadcastRecord{537d3428 android.net.wifi.RSSI_CHANGED}
  BroadcastRecord{537d4e14 com.android.providers.calendar.intent.CalendarProvider2}
  BroadcastRecord{537b91b8 android.net.conn.CONNECTIVITY_CHANGE}
  BroadcastRecord{537d3e54 android.intent.action.TIME_SET}
  BroadcastRecord{537cd3dc android.intent.action.ALARM_CHANGED}
  BroadcastRecord{5380b6a8 com.android.calendar.APPWIDGET_UPDATE}
  BroadcastRecord{53808e9c android.intent.action.PROVIDER_CHANGED}
  BroadcastRecord{538076a8 com.android.server.action.NETWORK_STATS_UPDATED}
```

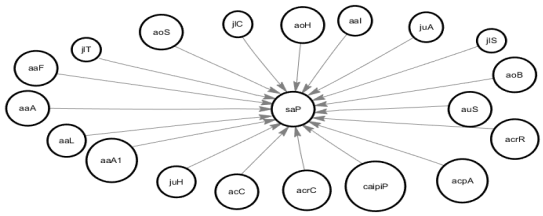
(a) Before Triggered

```
root@android:/ # pm list packages -3
  package:software.android
root@android:/ # dumpsys |grep BroadcastRecord
  BroadcastRecord{53addf2c com.android.server.action.NETWORK_STATS_UPDATED}
  BroadcastRecord{53b0894c SMS_SENT}
  BroadcastRecord{53864328 android.search.action.SEARCHABLES_CHANGED}
  BroadcastRecord{5387b7b0 android.intent.action.PACKAGE_ADDED}
  BroadcastRecord{538a8234 com.android.internal.telephony.gprs-data-stall}
  BroadcastRecord{53b38844 android.intent.action.CLOSE_SYSTEM_DIALOGS}
  BroadcastRecord{53920ef4 com.android.server.action.NETWORK_STATS_UPDATED}
```

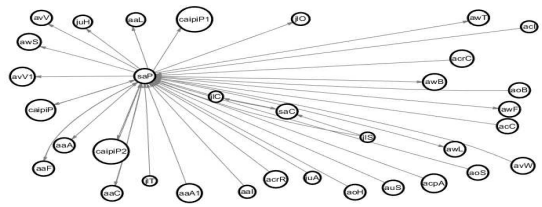
(b) After Triggered

Fig. 19. The Broadcasts in System

After the DAMBA captures these suspicious broadcasts in the background, the ORG extraction module is activated. In the first step of detection, we compare the static information of 'soft.apk' with the previously generated static base. We obtain the top five malicious families through static analysis and their possibilities are BaseBridge(159.32), Zsone(143.21), DroidDream(84.60), jSMSHider(72.18), AnserverBot(43.90). In the second step, we performed subgraph matching for these five families. The extracted ORG of 'soft.apk' is shown in Fig 20(b). Note, the disconnected nodes are not displayed and some class names are simplified to obtain a cleaner presentation. And the ORGB of BaseBridge is shown in Fig 20(a). Finally, only the BaseBridge in these five families matched successfully, which took 5.34 Seconds.



(a) The ORGB of BaseBridge Family



(b) The ORG of 'soft.apk'

Fig. 20. Dynamic Feature Graph (Partial)

V. Related Work

With the recent development in mobile security, a considerable number of studies have focusing on Android malware detection have been conducted [17]. In this section, we discuss previous studies related to object reference analysis, sub-graph isomorphism matching algorithm, and Android malicious behavior detection.

Object Reference Analysis

Chan et al. [6] propose dynamic birthmark system for Java and establish the ORG to detect code theft by using VFLib. However, they can only solve the application on PCs, and the objects they analyze can only be input independently. Furthermore, they lack the effective validation and optimization of time. Zhou et al. [46] adopt a traditional graph-based watermarking method, which searches through object reference relationships to detect a potential watermarking graph; the watermark recognizer calls the extended Dalvik virtual machine to export all object reference information in the heap which is too large to analyze efficiently and contains irrelevant relationships. In this study, we access memory objects in the heap and filter out useless information. We combine dynamic and static analyses to exploit the advantages of both. In addition, we propose the TANMAD algorithm, which is optimized for the Android environment to detect malicious apps by object reference analysis. The experimental results show that our method can effectively reduce unnecessary time overhead and improve accuracy.

Sub-graph Isomorphism Matching Algorithm

Researchers have proposed a number of algorithms for subgraph isomorphism matching. Proposed in 1976, the Ullmann algorithm [38] is still one of the most widely used homogeneous algorithms. This algorithm utilizes prediction equations to reduce the search space. Ghahraman et al. [20] propose an algorithm based on a monomorphism pattern; this algorithm

employs an association graph to reduce the searching space based on the net-graph matrix. However, the space cost of the matrix is high. Therefore, this algorithm is unsuitable for large-scale graph matching. The Nauty algorithm [26], which is based on group theory to solve the isomorphism problem, is the most rapid algorithm for isomorphism. It constructs an automorphism group for each graph, and then specification labels are established by the automorphism groups; the matching process analyzes the adjacent matrix of the labels to obtain the final result. In 2001, Cordella et al. [11] improve on the VF algorithm proposed in 1999, and calls their algorithm VF2; they adopt a heuristic search strategy and reduce the time complexity to $O(N)$; their experiment shows that the VF2 algorithm is applicable to isomorphism and sub-graph isomorphism problems.

Foggia et al. [18] conduct experimental tests on the aforementioned algorithms using multiple sets of graphs of different scales; their experiments reveal that VF2 algorithm and Nauty algorithm show improved results in dealing with randomly connected graphs similar to the ORG in our work. Among all the isomorphic patterns, only the isomorphism has not been proven to be an NPC problem, and thus, no universal polynomial time algorithm has been developed for the general problem [19]. Therefore, the time complexity of the isomorphism algorithm reaches $o(C^N)$ in the worst case. However, in a real-world scenario, the time cost is satisfactory in most cases mainly because of two reasons. First, the attributes of the edges and vertices can be used to optimize the running time of the algorithm. Second, the problem encountered in real apps is not the worst case. Owing to noise factors, the strictest isomorphic pattern is less used. To ensure the high matching accuracy, we adopt sub-graph isomorphism. The Nauty algorithm is only utilized for isomorphism matching; thus, we apply the fundamental strategy of the VF2 algorithm and propose the TANMAD algorithm. We optimize the detecting performance and accuracy by utilizing the Android static analysis and semantic pruning. To the best of our knowledge, the implementation of TANMAD is novel.

Android Malicious Behavior Detection

As mentioned in Section I, malicious behavior detection methods for the Android platform is generally divided into four types [37].

With the emergence of malware in the Android market, a number of methods is proposed to detect malicious apps based on static analysis. For instance, Woodpecker [21] analyzes the dangerously exposed permissions of preloaded apps to detect capability leaks in stock Android phones. IccTa [23] utilizes ICC analysis to reach a precise and complete identification of privacy leakage. DroidEagle [35], and MassVet [8] focus on detecting repackaged and clone attacks. DroidSIFT [42] establishes behavior graphs to describe the code logic, which is based on the source-code analysis. In general, static methods achieve high code coverage. Nevertheless, they lack real execution path and the associated execution context. Therefore, current malware hides malicious logic by utilizing code obfuscation technologies to evade the analysis based on static analysis. By contrast, we employ the ORGs extracted from heap memory to describe the feature of malware and can effectively resist obfuscation and dynamic code-loading attacks [30].

Dynamic analysis is also commonly adopted for Android malware detection. This technique is implemented by executing the test apps in virtual machines or the real smartphones. A number of studies have used dynamic analysis. Enck et al. [15] detect privacy leakage by tracking the transfer of sensitive information. Rastogi et al. [31] implement dynamic taint tracking and API kernel-level monitoring; they also apply an event trigger and use an intelligent execution technique to achieve high execution coverage. Zhang et al. [45] reconstruct the sensitive behaviors of Android apps by analyzing the required permissions dynamically. DroidScope [41] is based on the semantics of different levels to provide detailed information of malware behavior. Thus, dynamic detection, which requires a considerable amount of resources of users' smart devices and exhaustive inputs to trigger the malicious logic, is nearly impossible. A former work [39], aimed at detecting Android malware dynamically. To reduce the dependence on dynamic analysis, we optimize it from efficiency and accuracy by adopting both dynamic and static method. We pre-process suspicious apps statically to obtain a candidate list, thereby remarkably reducing the cost of dynamic analysis and improving detection accuracy. To improve the scalability of our method, we analyze the specific system activities caused by malware and develop the semiautomatic for script simulation operations. If there is a special input in the app, we take manual action to ensure that malicious code is called. Through the above methods, we greatly improved the scalability of DAMBA.

Considering the advantages of dynamic and static analysis, many previously proposed systems adopt both types of methods to complement each other, which is also called the hybrid detection. Zhou et al. [47] test the presence of malicious apps on five representative third-party Android app stores. They use static behavior analysis and dynamic heuristic detection to identify 0day malware. To guide dynamic analysis and extend the code coverage, Spreitzenbarth et al. [33] apply the static method to extract the permission required by an app and convert the Dalvik bytecode to smali code to search for dangerous features. However, this hybrid method encounters difficulty in determining whether a suspicious behavior is intended by the users. To deal with this problem, we use the ORGB to express the malicious feature as the ORGB can describe the similarities of malware apps and exclude accidentia. In addition, we observe the appropriate extracting occasion to ensure the malicious code is executed without the users' intention.

Machine learning-based detection has recently exhibited the most potential. Several detection systems that use learning algorithms have been proposed. MaMaDroid [29] extracts API invocation to construct Markov Chains of behavioral models, which are in turn used to extract the features for machine learning. Drebin [3] proposes a machine learning method based on static analysis and is directly used on Android devices. ICCDetector [40] systemically analyzed ICC

patterns of large-scale samples and outputs a trained model for detection. Marvin [24] combines static with dynamic analysis and use machine learning to classify malware. Overall, machine learning based technologies mainly focus on the accuracy of detection, but the performance and efficiency are not considered. And the quality of the dataset directly affects the training effect. By contrast, our work focuses on both performance and efficiency and has good scalability with a reasonable hardware cost.

VI. Conclusion and Future Work

Mobile malware is a rapidly growing threat. As a remedy, DAMBA is proposed in this study for Android malicious behavior detection. This system combines two approaches, namely, static and dynamic analyses, and exhibits the advantages of both. This system analyzes the trigger of the malicious code, and establishes the birthmark base by extracting malicious feature. In the matching process, the TANMAD algorithm, which includes two steps, is used. First, static analysis based on permissions and system classes is employed to filter out the families to which the test app is impossible to belong rapidly. Second, two pruning strategies and the parameter λ are employed to adjust the accuracy of different scales of the ORGB by applying the fundamental strategy of the VF2 algorithm.

In the experimental verification, 63% of irrelevant families are filtered out in the first step, and the top five families in the candidate list reach an accuracy of 95%. When the value of λ is 0.87, the matching result can reach both low false-negative and false-positive rates. In addition, to conserve the limited resources of the mobile client, the complex calculation task is completed by the PC client. The experiment results demonstrate that the mean time for detection is less than 5 s, and the shortest time is less than 1 s. Furthermore, our prototype can resist the code obfuscation and variant attacks.

Our work can be improved by considering more types of features, such as network flows and API calls, to enrich the birthmark base of the malware. Machine learning-based techniques can potentially be used to detect the never-before-seen attack. Furthermore, we will introduce the idea of parallelism to the sub-graph isomorphism algorithm to improve detection efficiency.

Acknowledgment

This work is supported in part by the National Key Research and Development Plan under grant No. 2016YFB0800801, the National Science Foundation of China (NSFC) under grant No. 61672186 and 61472108. (Corresponding author: Weizhe Zhang.)

References

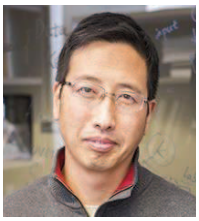
[1] Virustotal. https://www.virustotal.com/.
[2] Android malware genome project. http://www.malgenomeproject.org/.
[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.
[4] Baidu app market. http://shouji.baidu.com/.
[5] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.
[6] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. Dynamic software birthmark for java based on heap memory analysis. In *IFIP International Conference on Communications and Multimedia Security*, pages 94–107. Springer, 2011.
[7] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186. ACM, 2014.
[8] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security Symposium*, volume 15, 2015.
[9] Contagion mobile. http://contagiominidump.blogspot.com/.
[10] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the vf graph matching algorithm. In *Image Analysis and Processing, 1999. Proceedings. International Conference on*, pages 1172–1177. IEEE, 1999.
[11] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.
[12] Bart De Decker, Jorn Lapon, Vincent Naessens, and Andreas Uhl. *Communications and Multimedia Security: 12th IFIP TC 6/TC 11 International Conference, CMS 2011, Ghent, Belgium, October 19-21, 2011, Proceedings*, volume 7025. Springer, 2011.
[13] Android Developers. What is android. http://developer.android.com/guide/basics/what-is-android.html, 2011.
[14] Android Developers. App manifest file. http://developer.android.com/guide/topics/manifest/manifest-intro.html, 2012.
[15] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.
[16] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.
[17] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 12(8):1772–1785, 2017.
[18] Pasquale Foggia, Carlo Sansone, and Mario Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.
[19] Scott Fortin. The graph isomorphism problem. 1996.
[20] David E Ghahraman, Andrew KC Wong, and Tung Au. Graph optimal monomorphism algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(4):181–188, 1980.
[21] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, volume 14, page 19, 2012.
[22] K Hong. Android 4 13. intent. http://bogotobogo.com/Android/android13Intent.php, 2016.
[23] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.
[24] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 422–433. IEEE, 2015.
[25] McAfee. Mcafee mobile security. https://pccw.mcafeemobilesecurity.com/, 2018.
[26] Brendan D McKay et al. Practical graph isomorphism. 1981.
[27] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[28] Ginger Myles and Christian Collberg. Detecting software theft via whole program path birthmarks. In *International Conference on Information Security*, pages 404–415. Springer, 2004.
[29] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *arXiv preprint arXiv:1711.07477*, 2017.
[30] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.
[31] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.
[32] Ursula Redmond and Pádraig Cunningham. Identifying over-represented temporal processes in complex networks. *Dynamic Networks and Knowledge Discovery*, page 61, 2014.
[33] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.
[34] Statista. Smartphone os market share. https://www.statista.com/statistics/266219/global-smartphone-sales-since-1st-quarter-2009-by-operating-system/, 2017.
[35] Mingshen Sun, Mengmeng Li, and John Lui. Droideagle: seamless detection of visually similar android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 9. ACM, 2015.
[36] Symantec. Android basebridge. https://www.symantec.com/security-center/writeup/2011-060915-4938-99.
[37] Darell JJ Tan, Tong-Wei Chua, Vrizlynn LL Thing, et al. Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 47(4):58, 2015.
[38] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.
[39] Huanran Wang, Hui He, and Weizhe Zhang. Demadroid: Object reference graph-based malware detection in android. *Security and Communication Networks*, 2018, 2018.
[40] Ke Xu, Yingjiu Li, and Robert H Deng. Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264, 2016.
[41] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European symposium on research in computer security*, pages 163–182. Springer, 2014.
[42] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.
[43] Weizhe Zhang, Hui He, Qizhen Zhang, and Tai-hoon Kim. Phoneprotector: protecting user privacy on the android-based mobile platform. *International Journal of Distributed Sensor Networks*, 10(2):282417, 2014.
[44] Weizhe Zhang, Xiong Li, Naixue Xiong, and Athanasios V Vasilakos. Android platform-based individual privacy information protection system. *Personal and Ubiquitous Computing*, 20(6):875–884, 2016.
[45] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.
[46] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12. ACM, 2013.
[47] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.

Weizhe Zhang is currently a professor in the School of Computer Science and Technology at Harbin Institute of Technology, Harbin, China, and director in the Cyberspace Security Research Center, Pengcheng Laboratory, Shenzhen, China. His research interests are primarily in parallel computing, distributed computing, cloud and grid computing, and computer network. He has published more than 100 academic papers in journals, books, and conference proceedings. He is a member of the IEEE.

Huanran Wang received the BS degree in software engineering from Harbin Engineering University, China, in 2016. He is currently working toward the PhD degree in the School of Computer Science and Technology, Harbin Institute of Technology. His research interests include Mobile Security, secure Internet of Things, and Cyberspace Security.

Hui He is currently a full professor of network security center in the Department of Computer Science, China. She received the Ph.D. in department of computer science from the Harbin Institute of Technology, China. Her research interests are mainly focused on distributed computing, IoT and big data analysis. She is a member of the IEEE.



Peng Liu received his BS and MS degrees from the University of Science and Technology of China, and his PhD from George Mason University in 1999. Dr. Liu is a Professor of Information Sciences and Technology, founding director of the Center for Cyber-Security, Information Privacy, and Trust, and founding director of the Cyber Security Lab at Penn State University. His research interests are in all areas of computer and network security. He has published a monograph and over 260 refereed technical papers. His research has been sponsored by NSF, ARO, AFOSR, DARPA, DHS, DOE, AFRL, NSA, TTC, CISCO, and HP. He has served on over 100 program committees and reviewed papers for numerous journals. He is a recipient of the DOE Early Career Principle Investigator Award. He has co-led the effort to make Penn State a NSA-certified National Center of Excellence in Information Assurance Education and Research. He has advised or co-advised over 30 PhD dissertations to completion.