Hindawi
Mathematical Problems in Engineering
Volume 2020, Article ID 3842094, 15 pages
https://doi.org/10.1155/2020/3842094

Research Article

A Graph-Based Feature Generation Approach in Android Malware Detection with Machine Learning Techniques

Xiaojian Liu, Qian Lei, and Kehong Liu

School of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an, Shaanxi 710054, China

Correspondence should be addressed to Xiaojian Liu; 780209965@qq.com

Received 20 October 2019; Revised 31 March 2020; Accepted 7 April 2020; Published 27 May 2020

Academic Editor: Eric Florentin

An explosive spread of Android malware causes a serious concern for Android application security. One of the solutions to detecting malicious payloads sneaking in an application is to treat the detection as a *binary classification* problem, which can be effectively tackled with traditional machine learning techniques. The key factors in detecting Android malware with machine learning techniques are *feature selection* and *generation*. Most of the existing approaches select and generate features without fully examining the structures of programs, and thus the important semantic information associated with these features is lost, consequently resulting in a low accuracy rate in detection. To address this issue, we propose a new feature generation approach for Android applications, which takes components and program structures into consideration and extracts features in a graph-based and semantics-rich style. This approach highlights two major distinguishing aspects: the *context-based feature selection* and *graph-based feature generation*. We abstract an Android application as a collection of reduced iCFGs (interprocedural control flow graphs) and extract original features from these graphs. Combining the original features and their contexts together, we generate new features which hold richer semantic information than the original ones. By embedding the features into a feature vector space, we can use machine learning techniques to train a malware detector. The experiment results show that this approach achieves an accuracy rate of 95.4% and a recall rate of 96.5%, which prove the effectiveness and advantages of our approach.

1. Introduction

Android system, as one of the most popular mobile platforms, faces various serious security challenges due to its open-source characteristics, imperfect permission mechanisms, and the absence of full certification of applications at their publications. Malware or malicious payload exploits the vulnerabilities in Android system to implement a variety of attacks, such as privilege escalation, remote control, illegal financial charges, and personal information stealing, resulting in privacy leakage and even serious financial losses. Therefore, there is an imperative need to detect malicious payload and analyze its potential impact when installing and using an Android application.

Machine learning techniques have been widely used in malware detection [1–8]. This kind of approach treats malware detection as a binary classification problem (i.e., differentiate an application as malicious or benign), which can be tackled with the traditional techniques in pattern recognition or machine learning disciplines. Since this approach does not fully investigate the semantics and all details of programs, it achieves a better performance over traditional *dynamic approaches* [1, 2, 9] and *static approaches* [4–6, 10–15] in terms of scalability and time consumption.

The key factors affecting the performance of the machine learning-based approaches are feature selection and generation. APIs and permissions [1, 3] are commonly selected as the features in that they hold rich security-related information about what critical resources can be accessed by which operations. However, in most of the existing works, these features are extracted in a level of whole-application granularity, and their associated contexts are neglected, resulting in a high false-positive ratio in detection.

To tackle this problem, we propose a static detection method for Android malware, which improves the existing works by additionally considering the contexts of features.

We identify three kinds of program characteristics as raw features: *security-sensitive broadcast events*, *security-sensitive permissions*, and *bigrams of API calls*, each kind of which associates with a kind of context. We combine these raw features with their contexts to generate new features, which are subsequently used to train and test a classifier model by embedding them into a feature vector space.

The main challenge of our approach is to retrieve the features and their contexts from original program codes. To achieve a better performance in terms of time consumption, we adopt a graph-based technique to the feature generation. We define the structure of a program as a set of iCFGs and offer a graph reduction transformation for the iCFGs to simplify their structures and improve the performance of feature generation. Based on the reduced iCFGs, we interpret the set of their edges as the set of bigrams of API calls involved in callbacks, which allows us to design an efficient graph algorithm to extract the features.

In summary, the main contributions of this paper are as follows:

(1) We propose a context-based feature selection approach, which combines the three kinds of raw features with their contexts to serve as newly generated features. Since these features hold rich semantic information about program behaviors, we achieve a better result than the traditional machine learning-based approaches.

(2) We propose a graph-based feature generation approach. Since we only concern with the APIs to be called, we can safely remove irrelevant graph nodes and edges and reduce the complex structure of an iCFG to a simplified version. In addition, we establish a direct mapping from the edges of a reduced iCFG to the bigrams of API calls, which leads to an efficient algorithm for the feature generation.

(3) With the newly generated features, we trained a classifier model using several state-of-the-art machine learning algorithms on 4972 samples in total, 3732 for training and 1240 for testing. The comparison experiments show that the random forest algorithm has the best performance on the selected feature set, with an accuracy rate of 95.4% and a recall rate of 96.5%, which prove the effectiveness of our approach.

The remainder of this paper is organized as follows. Section 2 describes the selected features in Android malware detection. The process of generating these features from an Android application is presented in Section 3. Section 4 focuses on transforming the features into values and embedding the features into a feature vector space. We implement our approach and report the experiment results in Section 5. Finally, we represent related works in Section 6 and conclusion in Section 7.

2. Feature Selection

Since API calls, permissions, and system broadcast events usually hold rich security-related information, they are commonly selected as features in traditional malware detection approaches. In this paper, we call these features as *original features* or *raw features* and combine them with their contexts together to form newly generated features to achieve better detection results. For convenience, we simply call these new features as *features* when no confusion is possible. In what follows, we first introduce the original features we selected and then associate different contexts with them to form the features.

2.1. Raw Features. The approach makes use of three categories of information as raw features:

(i) Bigrams of security-sensitive API calls

(ii) Sensitive permissions

(iii) Sensitive broadcast events

2.1.1. Bigrams of API Calls. The N-gram model [3] is a language model widely used for the NLP (natural language processing), such as large vocabulary continuous speech recognition, machine translation, and text classification. The N-gram model is based on the assumption that the appearance of a word is only related to the previous $N-1$ words. Moving a sliding window of fixed length N from the beginning to the end of a text will generate a number of N-grams of words each of which contains N sequential words. For example, the division of "I am a citizen of the People's Republic of China" forms a set of bigrams: {I am, am a, a citizen, citizen of, of the, the People's, People's Republic, Republic of, of China}.

If we consider the generalized-sensitive APIs defined in RepassDroid [11] as the vocabulary, then an N-gram of words is just a sequence of N consecutive API calls. Considering a large value of N will sharply increase the burden of computation; in most cases, N is set to 2 or 3. Here we adopt $N = 2$, i.e., bigrams of API calls, as one kind of features.

We analyze 1,000 typical malicious applications and then extract all bigrams of API calls from them and calculate the frequency of each bigram. In order to ensure the accuracy and efficiency of the detection, we finally choose 300 bigrams of API calls that appear most frequently in malicious applications as the raw features. A part of these bigrams of API calls is shown in Table 1.

2.1.2. Sensitive Permissions. To select a proper number of permissions as the raw features, we borrow the statistical indicator TF-IDF [16] (term frequency-inverse document frequency) from a field of text mining to measure the importance of a permission with respect to a program.

We employ a TF-IDF-like approach [17] to select 20 sensitive permissions that can effectively distinguish malicious applications from benign ones. First, we divide the corpus into two categories: *malicious* corpus ($k = 1$) and *benign* corpus ($k = 2$), and then define three metrics for each permission s in category k ($k = 1, 2$) to characterize its usage in different categories:

Table 1: A part of the bigrams of API calls.

Bigrams of API calls	Semantics of bigrams	Frequency (%)
getDeviceId(); getSubscriberId()	Read the IMEI and IMSI values of the device	83.3
getSubscriberId(); sendTextMessage()	Get IMSI value of sim card and send it via SMS	76.8
openConnection(); getOutputStream()	Connect to the network and send data to the server	58.6
getDisplayOriginatingAddress(); getDisplayMessageBody()	Get the sender's address information and information content	56.7
getSystemService("alarm"); getBroadcast()	Get the alarm service and take a broadcast as its response	43.6
getOriginatingAddress(); getMessageBody()	Intercept SMS address and SMS content	43.3
getSystemService(CONNECTIVITY_SERVICE); getActiveNetworkInfo()	Determine the network status	39.6
getLongitude(); getLatitude()	Get user's geographic location	31.5
getSubscriberId(); getSimSerialNumber()	Obtain IMEI ICCID values of the device	23.4
Query(); getString()	Query the database and get the query results	21.2

num (s,k): number of samples that use permission s in category k.

$\text{per}(s,k) = \text{num}(s,k)/\text{falNumber}(k)$: percentage of samples that use permission s in category k, where falNumber (k) denotes the number of samples in k.

$w(s,k)$: TF-IDF value of permission s in category k.

In addition, we use allnumber to denote the number of all collected samples and totalNumber(s) to denote the number of the samples that use permission s in both categories, i.e.,

$$\text{totalNumber}(s) = \sum_{k=1,2} \text{num}(s,k). \quad (1)$$

According to the above definition, the value of $w(s, k)$ should be positively related with its *per* (s, k) and be negatively related with its *totalNumber* (s). Then, $w(s, k)$ of permission s in category k is defined as follows:

$$w(s,k) = \text{per}(s,k) \times \log_2 \frac{\text{allNumber}}{\text{totalNumber}(s)}. \quad (2)$$

We collect 150 malicious samples and 150 benign samples to calculate $w(s,k)$. Finally, we select the top 20 permissions with the highest $w(s,k)$ as the raw permission features, as shown in Table 2.

Table 2: Sensitive permissions and their $w(s,k)$ values.

Sensitive permission (s)	w (s, $k=1$)
SEND_SMS	0.243
READ_PHONE_STATE	0.239
READ_CALL_LOG	0.183
RECEIVE_SMS	0.182
READ_EXTERNAL_STORAGE	0.177
READ_CONTACTS	0.172
RECEIVE_BOOT_COMPLETED	0.171
CALL_PHONE	0.169
INSTALL_PACKAGE	0.167
ACCESS_WIFI_STATE	0.166
WRITE_EXTRNAL_STORAGE	0.165
MOUNT_UNMOUNT_FILESYSTEMS	0.165
READ_SMS	0.163
ACCESS_NETWORK_STATE	0.158
PROCESS_OUTCALLS	0.149
CHANGE_WIFI_STATE	0.148
ACCESS_FINE_LOCATION	0.144
ACCESS_COARSE_LOCATION	0.140
INTERNET	0.140
WRITE_SETTING	0.091

2.1.3. Sensitive System Broadcasts. Android malware usually relies on system broadcast events and their corresponding callbacks to trigger malicious payload [18]. For example, BOOT_COMPLETED is a broadcast event sent by the Android system whenever it finishes its booting process; some malware may listen to this event and trigger malicious payloads or kick off the background services after it occurs. Work [18] collects 25 most related events in Android malware, and we take them as the raw features of broadcast events.

2.2. Contexts of Raw Features. Each kind of the raw feature is associated with a kind of context. We prepare for our formal description by first giving the definitions of the following sets:

BIGRAMS: the finite set of all bigrams of API calls

PERMS: the finite set of sensitive permissions

EVENTS: the finite set of sensitive broadcast events

APPS: the finite set of applications

COMPS: the finite set of components defined in applications

CALLBACKS: the finite set of callback functions defined in components

Definition 1 (contexts of the raw features). For a raw feature of bigram of API calls, its context is defined as a function *context_bigram*:

$$context\_bigram: \text{BIGRAMS} \longrightarrow 2^{\text{CALLBACKS}}$$

Here, $2^{\text{CALLBACKS}}$ denotes the powerset of CALLBACKS. Likewise, the context for a feature of permission or broadcast event is defined as follows:

$$context\_perm: \text{PERMS} \longrightarrow 2^{\text{COMPS}}$$

$$context\_event: \text{EVENTS} \longrightarrow \text{APPS}$$

For a bigram feature $b \in$ BIGRAMS, its context is a set of callback functions that invoke the sequence of APIs of b; for a permission feature $p \in$ PERMS, its context is a set of components using p as one of its actually used permissions; and for an event feature $e \in$ EVENTS, its context is just the application which listens to e and triggers some behavior whenever e is received.

Definition 2 (features). A feature of an Android application is defined as a pair (f, c), where $f \in$ BIGRAMS$\cup$PERMS$\cup$ EVENTS is a raw feature and $c = context\_*(f)$ is the context of f.

Note that the context of a raw feature may comprise more than one element. A bigram of API calls may be invoked in multiple callback functions, and thus its context should include all of the callbacks calling it; a permission may be used in multiple components, and thus its context should comprise all of these components. For example, if the bigram getDeviceId (); getSubscriberId () is invoked by both OnReceiver () and OnClick (), then its context is {OnReceive (), OnClick ()}, and the feature is written:

(getDeviceId (); getSubscriberId (), {OnReceive (), OnClick ()}).

3. Feature Generation

In this section, we introduce how to generate features from an Android application. Generating features is far from simply parsing program texts; instead, it needs a comprehensive analysis for program structures. We first outline the process of feature generation and then focus on a graph model of program structures and a graph reduction transformation for iCFGs.

3.1. Outline of Feature Generation Process. The sensitive broadcast events are usually configured statically in an application's *manifest.xml* file, and thus they can be directly captured from the manifest file. However, extracting permissions is not straightforward, although they are also configured in manifest. The main difficulties come from below two factors: first, the permissions declared in *manifest.xml* are required for a level of whole-application granularity rather than for individual components; we thus cannot directly acquire the contexts of the permissions; moreover, the declared permissions in *manifest.xml* are not always identical to the permissions really used by the application. It is very common that an application always tends to require more permissions than it truly needs. In short, simply capturing permissions from *manifest.xml* does not work well for the generation of permission features.

Our approach extracts both sensitive permissions and the bigrams of API calls in the same process via a comprehensive analysis of programs. The complete process of feature generation is illustrated in Figure 1.

3.2. Graph-Based Feature Generation. The critical step in previous feature generation process is to construct an iCFG for a callback function. In this section, we first give definitions for CGs, CFGs, and iCFGs, then propose a graph reduction transformation rule to simplify an iCFG's structure, and finally give a theorem that states how to extract bigrams of API calls from the reduced iCFG.

3.2.1. CG, CFG, and iCFG

Definition 3 (call graph). A call graph of an application a is defined as a labeled multigraph:

$$\text{CG}_a = (N, S, E, r), \tag{3}$$

where

(i) N is a finite set of nodes, and each node is labeled a class method name in the set $\{m, m_1, m_2, \ldots, m_k\}$. The leaf nodes of CG_a represent API calls.

(ii) S is a finite set of call-site labels, and each label represents a statement of method call.

(iii) $E \subseteq N \times S \times N$ is the finite set of labeled directed edges. Each edge $e = (m_i, s_k, m_j) \in E$ denotes that class method m_i calls class method m_j at call site s_k.

(iv) $r \in N$ is the unique entry node of CG_a, representing the root class method m.

Note that as a class method may be called at different call sites, the call graph is actually a multigraph with multiple directed edges from one node to another or alternatively multiple labels on some edges.

A call graph merely reveals a hierarchy of class method calls but cannot further specify in what sequence these methods are invoked. To describe the complete internal behavior of a class method, we need to introduce the control flow graph for a class method, which specifies the control flow transfers and the sequence of method calls (including API calls) within the class method.

Definition 4 (control flow graph). A CFG of a method m is defined as a directed simple graph

$$\text{CFG}_m = (N, E, s, e), \tag{4}$$

where

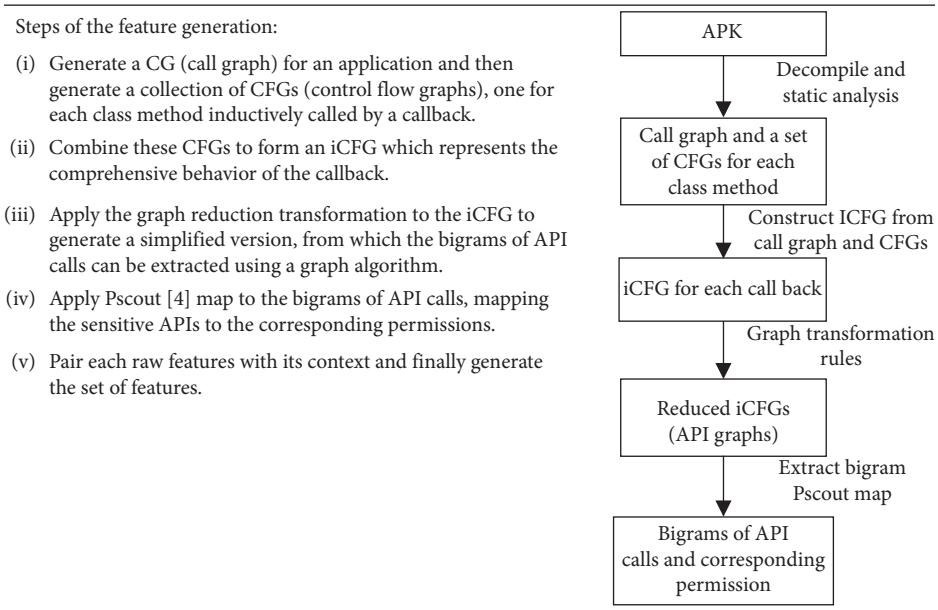

Figure 1: The process of feature generation.

(i) N is a finite set of nodes. Each node represents a statement in the set $\{s_1, s_2, \ldots, s_n\}$. A statement can be an assignment, conditional branch, method call, or call return (including API call or return statement).

(ii) $E \subseteq N \times N$ is the finite set of edges. Each edge $e = (n_i, n_j) \in E$ indicates a control flow transfer from node n_i to node n_j.

(iii) $s \in N$ is the unique *start* node, representing the control flow starts to enter the method m.

(iv) $e \in N$ is the unique *exit* node, representing the control flow exits from the method m.

As a class method may invoke other class methods (may also call itself to form recursions) in its body, we need to construct a CFG to describe the control flow transfers at the points of method calls and call returns; this kind of graph is called an interprocedural CFG, or iCFG for short. An iCFG is a supergraph that integrates a number of individual CFGs to form a holistic one.

Given a callback function m_0, let $\{m_1, m_2, \ldots, m_k\}$ be the set of class methods called by m_0, and $\mathrm{CFG}_i = (N_i, E_i, s_i, e_i)$ be the control flow graphs of m_i, $i \in \{0, 1, 2, \ldots, k\}$. Every method invocation in m_i contributes two nodes: a ***call*** node and a ***return*** node. Let $Call_i \subseteq N_i$, $Return_i \subseteq N_i$ be the sets of call nodes and return nodes of CFG_i, respectively.

CFG_i's edges are divided into two disjoint subsets: $E_i = E_i^0 \cup E_i^1$, where an edge $(n_1, n_2) \in E_i^0$ is an ordinary control flow edge—it represents a direct transfer of control from a node to another; an edge $(n_1, n_2) \in E_i^1$ if n_1 is a call node and n_2 is the corresponding return node.

Definition 5. (interprocedural CFG). An iCFG for a callback function m_0 is a supergraph:

$$iCFG_{m0}^{*} = \left(N^{*}, E^{*}, s_0, e_0\right), \tag{5}$$

where

(i) $N^{*} = \cup_{i \in \{0,1,2,\ldots,k\}} N_i$ is a finite set of nodes.

(ii) $E^{*} = E^0 \cup E^1 \cup E^2$, in which $E^0 = \cup_{\in \{0,1,2,\ldots,k\}} E_i^0$ is the collection of all ordinary control-flow edges; $E^1 = \cup_{i \in \{0,1,2,\ldots,k\}} E_i^1$ is the collection of all edges from call nodes to the corresponding return nodes; and E^2 is the set of ***call*** edges or ***return*** edges. An edge $(n_1, n_2) \in E^2$ is a call edge if n_1 is a call node and n_2 is the start node of the called method; an edge $(n_1, n_2) \in E^2$ is a return edge if n_1 is an exit node of some method m and n_2 is a return node immediately following a call to m. A call edge (n_1, s_i) and a return edge (e_j, n_2) correspond to each other if $i = j$ and $(n_1, n_2) \in E^1$.

(iii) s_0 is the entry node of callback m_0.

(iv) e_0 is the exit node of m_0.

For simplicity, we use *Call*, *Return*, *Start*, and *Exit* to denote the sets of all call nodes, return nodes, entry nodes, and exit nodes in an iCFG, respectively, and collectively call them structural nodes:

$$Call = \cup_{i \in \{0,1,2,\ldots,k\}} Call_i$$
$$Return = \cup_{i \in \{0,1,2,\ldots,k\}} Return_i$$
$$Start = \{s_0, s_1, \ldots, s_k\}$$
$$Exit = \{e_0, e_1, \ldots, e_k\}$$
$$StructNodes = Call \cup Return \cup Start \cup Exit$$

3.2.2. Reduction of iCFGs. To generate the features from iCFGs, we only need to concentrate on these nodes that represent the security-sensitive API calls; that is, we shall

reduce an original iCFG to a simplified version by removing the nodes and edges without contribution to the concerned features. Before introducing the reduction transformation, we shall first review some helpful terminologies in the following.

Given a general-purpose simple graph $G = (N, E)$, where N is the set of nodes and E is the set of edges, a path from node u to v in G, denoted as (u, v)-*path*, is a sequence of nodes:

$$(x_0 = u, x_1, x_2, \ldots, x_n = v), \tag{6}$$

where $(x_i, x_{i+1}) \in E$, for $i = 0, 1, \ldots, n-1$. In this case, we call v is reachable from u via the (u, v)-*path* or the path *passes through* or *traverses* the nodes $x_1, x_2, \ldots, x_{n-1}$ from u to v.

Assume $N' \subseteq N$ is a subset of N; a (u, v)-path of G is said to *traverse* N' from u to v, if $u, v \notin N'$ and $x_i \in N'$ for $i = 1, \ldots, n-1$. Note that (u, v)-path may be a circuit (or cycle) if it begins and ends at the same node, that is, $u = v$.

Figure 2 shows a graph G, in which $N = \{u, x_1, x_2, x_3, x_4, v\}$ and $N' = \{x_1, x_2, x_3\}$ whose nodes are shown as the shaded. There are three (u, v)-paths: (u, x_1, x_2, x_3, v), (u, x_1, x_2, x_4, v), and (u, x_4, v), in which only the path (u, x_1, x_2, x_3, v) traverses N' from u to v.

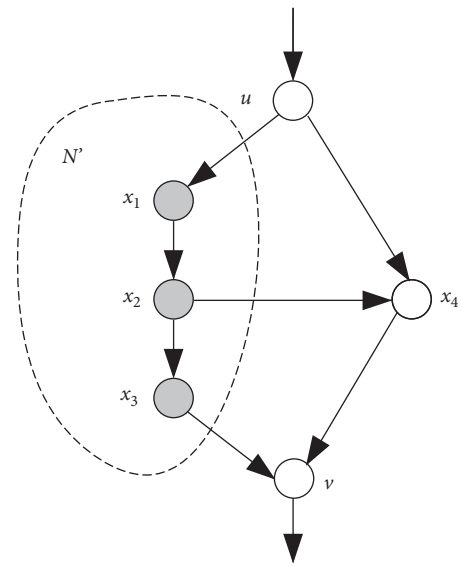

Figure 2: Path (u, x_1, x_2, x_3, v) traversing N'.

Definition 6 (reduction of iCFG). Let G be a finite set of iCFGs. The graph reduction transformation for iCFGs is a function $\boldsymbol{T}: G \longrightarrow G$ that transforms an iCFG $g = (N^*, E^*, s_0, e_0) \in G$ to a reduced iCFG $g' = (N^{*'}, E^{*'}, s_0', e_0')$:

$$\mathbf{T}(g) = g', \tag{7}$$

where

(i) $N^{*'} \subseteq N^*$ is the set of nodes including only *StructNodes* and the nodes representing the sensitive API calls in N^*

(ii) $E^{*'} = \{(n_1, n_2) \in N^{*'} \times N^{*'} \mid (n_1, n_2) \in E^*\} \cup \{(n_1, n_2) \in N^{*'} \times N^{*'} \mid (n_1, n_2) \notin E^* \wedge \exists (n_1, n_2)\text{-path such that it traverses } N^* - N^{*'} \text{ from } n_1 \text{ to } n_2 \text{ in } g\}$

(iii) $s_0' = s_0$

(iv) $e_0' = e_0$

For convenience, we refer to the reduced iCFGs as *API graphs*. From above definition, we can conclude the following properties about the reduction transformation.

Property 1. Critical nodes are preserved: all of the sensitive API call nodes and the structural nodes (i.e., the entry, exit, call, and return nodes) in g are preserved in g'.

Property 2. Reachability is preserved: for any $n_1, n_2 \in N^{*'}$, if n_2 is reachable from n_1 in g, then n_2 is also reachable from n_1 in g'.

We can easily observe that each ordinary directed edge $(n_1, n_2) \in E^0$ of an API graph actually represents a bigram of API calls $l(n_1); l(n_2)$, where $l(n_1)$ and $l(n_2)$ represent the sensitive API calls associated with n_1 and n_2, respectively. This observation indicates that we can calculate the set of bigrams of API calls from edges of API graphs, just as the following theorem states.

Theorem 1. *Let $g = (N^*, E^*, s_0, e_0)$ be an API graph for a callback method m_0, n_1 and n_2 be two nodes in N^*, and BIGRAM be the set of bigrams of API calls appearing in m_0. Then, BIGRAM can be calculated using the following equation:*

$$BIGRAM = \{l(n_1); l(n_2) \mid (n_1, n_2) \in E^0\} \cup \{l(n_1); l(n_2) \mid \exists p \in Call, q \in Start \bullet (n_1, p) E^0 \wedge (p, q) \in E^2 \wedge (q, n_2) \in E^0\} \cup \{l(n_1); l(n_2) \mid \exists p \in Exit, q \in Return \bullet (n_1, p) \in E^0 \wedge (p, q) \in E^2 \wedge (q, n_2) \in E^0\}.$$

Example 1. To illustrate the reduction transformation of iCFGs, we give a realistic example. In Table 3, the callback function OnClick () calls both the method method1 () and some APIs; Figure 3 shows how to reduce the iCFG of OnClick () to the corresponding API graph and then generate the bigrams of API calls from the resulting API graph. After obtaining the *BIGRAM*, we can easily obtain the permissions through the PScout mapping.

4. Feature Transformation

In order to utilize machine learning techniques to train a malware classification model, we need to transform the generated features into a feature vector space. Each feature vector represents the set of features for an application sample.

4.1. Feature Vectors. We define a feature vector as an array of 345 elements, as shown in Figure 4. We select 300 bigrams of API calls (Table 1 gives a part of the bigrams), 20 permissions (shown in Table 2), and 25 system broadcast events as the prominent features in the feature vector. An index i $(1 \le i \le 345)$ of the vector corresponds to a fixed raw feature, and the ith element's value $Vector[i]$ represents the quantitative value induced from this raw feature. For example, the first element of the vector corresponds to the raw feature of the bigram of API calls

Table 3: A brief Smali snippet code of OnClick () callback.

Proc OnClick :	
1	$r0: = @this
2	$r1 = com.example.leakageapp.Test1.method1 ()
3	$r2 = android.telephony.SmsManager.getDefault ()
4	$i0 = 0
5	label1: if $i0 < 3 goto label2
6	$r2.sendTextMessage("+4402073210905," null," $r1, null, null);
7	$i0 = $i0 + 1
8	goto label1
9	return
Proc method1 :	
10	$r0: = @this;
11	$r2 = new android.telephony.TelephonyManager;
12	$r3 = $r2.getDeviceId ();
13	$r4 = $r2.getSubScriberId ();
14	$r5 = $r3 + $r4;
15	return $r5;

getDeviceId (); getSubscriberId (), and $Vector[1]$, i.e., a_1, represents the value calculated from this feature.

Let f_i be the ith raw feature in the vector; its context is $c_i = context\_*(f_i)$, and thus the feature corresponding to f_i is (f_i, c_i) (see Definition 2). The value of the feature (f_i, c_i), namely, $Vector[i]$, can be calculated by applying a function f to (f_i, c_i):

$$Vector[i] = f(f_i, c_i).$$

Here, f is referred to as a *feature transformation function*, simply *feature function*, which presents different forms depending on the kinds of the raw features; the values of $f(f_i, c_i)$ are called the *feature values* of (f_i, c_i). The definitions of the feature functions are described in the following section.

4.2. Feature Functions

4.2.1. Feature Function for Bigrams

Definition 7. (feature function for bigrams). Let $b \in$ BIGRAMS be a raw feature of bigrams; it constitutes a feature $(b, context\_bigram\ (b))$. The feature function on this feature is defined as

$$\begin{cases} f(b, context\_bigram(b)) = \sum^{|UI|} w_{11} + \sum^{|Non\_UI|} w_{12}, & \text{if } context\_bigram(b) \neq \varnothing, \\ f(b, context\_bigram(b)) = 0, & \text{if } context\_bigram(b) = \varnothing, \end{cases} \tag{8}$$

where

(i) **UI** and **Non_UI** are disjoint subsets of $context\_bigram\ (b)$ and UI ∪ Non_UI = $context\_bigram\ (b)$. UI represents the set of the callbacks related to user interface operations, such as OnClick () and onPress (); likewise, Non_UI represents the set of the callbacks unrelated to user interfaces. |UI| is the cardinality of set UI.

(ii) w_{1i} $(i = 1, 2)$ are the weights associated with the callbacks in $context\_bigram\ (b)$. w_{11} is the weight of the UI-related callbacks; w_{12} the weight of the Non_UI-related callbacks. $\sum^{|UI|} w_{11} + \sum^{Non\_UI} w_{12}$ computes the sum of the weights associated with each of the callbacks in the context of b.

The values of w_{1i} are calculated using a statistical method. We first take 1000 malicious applications and 1000 benign ones as samples and then count the average frequency of the bigrams' occurrences in UI-related callbacks and Non_UI-related callbacks. The statistical results are shown in Table 4.

According to Table 4, we calculate the frequency difference of the bigram features in different contexts by the following formula:

$$\phi(c) = \frac{C_m^2(c)}{C_n(c)}, \tag{9}$$

where $C_m\ (c)$ is the frequency of bigrams with context c in malicious applications and $C_n\ (c)$ is the one with context c in benign applications. Table 5 gives the values of $\varphi\ (c)$ of each context. For convenience, $\varphi\ (c)$ is normalized as $\varphi'\ (c)$, which is assigned to the weight w_{1i}, i.e., w_{11} takes the value of $\varphi'(\text{UI}) = 0.14$ and w_{12} takes the value of $\varphi\prime(\text{Non\_UI}) = 0.86$.

4.2.2. Feature Function for Sensitive Permissions.

The contexts of sensitive permissions are divided into four categories according to the component types: *Activity*, *Service*, *Receiver*, and *Provider*. The feature function is defined as follows.

Definition 8. (feature function for permissions). Let $p \in$ PERMS be a raw feature of sensitive permission; it constitutes a feature $(p, context\_perm\ (p))$. The feature function for this feature is defined as

$$\begin{cases} f(p, context\_perm(p)) = \sum^{|setActivity|} w_{21} + \sum^{|setService|} w_{22} + \sum^{|setReceiver|} w_{23} + \sum^{|setProvider|} w_{24}, & \text{if } context\_perm(p) \neq \varnothing, \\ f(p, context\_perm(p)) = 0, & \text{if } context\_perm(p) = \varnothing, \end{cases} \tag{10}$$

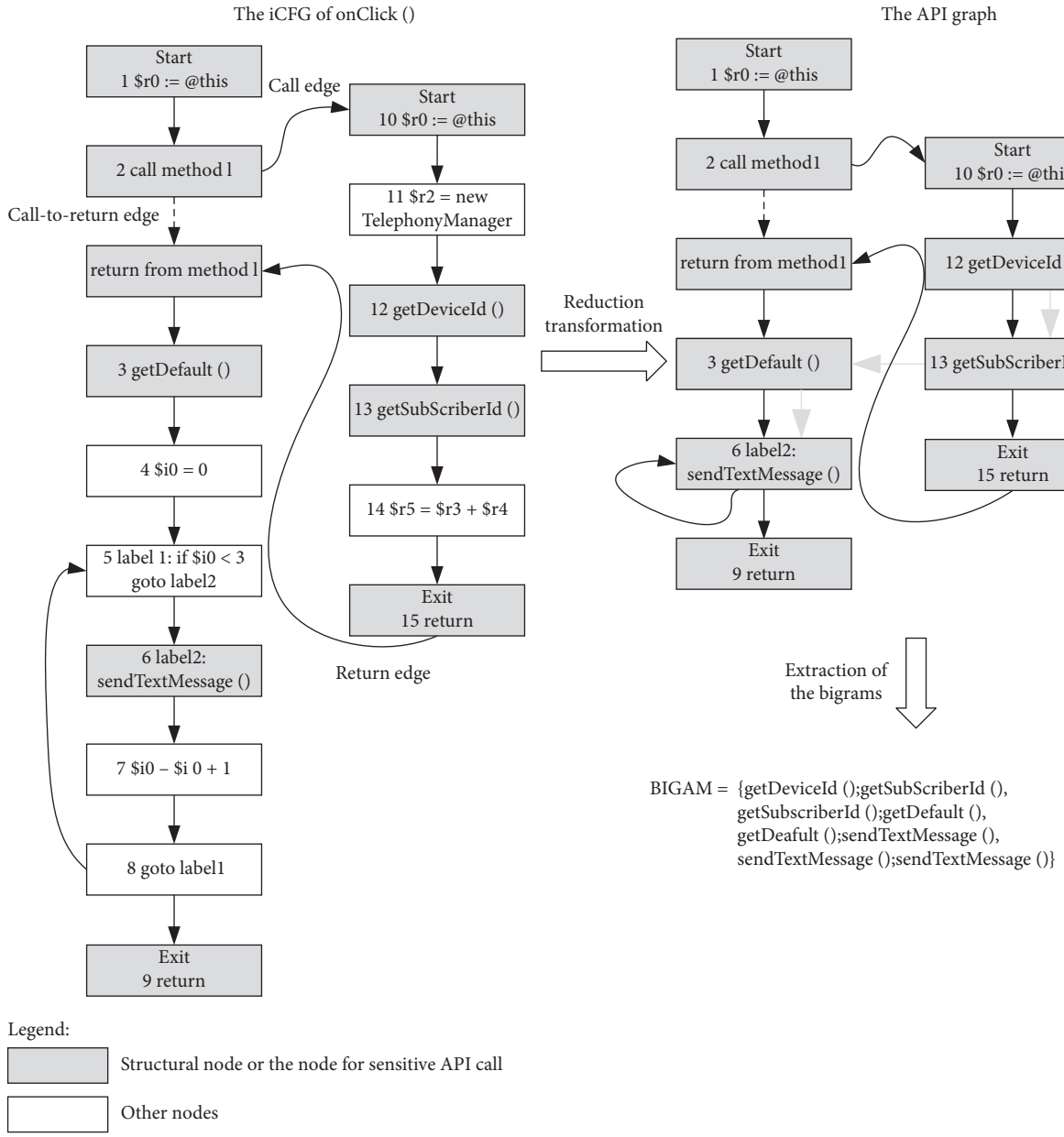

Figure 3: Transformation of OnClick ()'s iCFG to the API graph and extraction of *BIGRAM*.

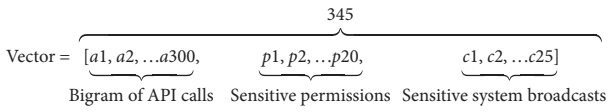

Figure 4: Feature vector.

where

(i) **SetActivity**, **SetService**, **SetReceiver**, and **SetProvider** are mutually exclusive subsets of *context_perm* (*p*). For example, SetActivity is the set of activity components in *context_perm* (*p*). |Set| is the cardinality of Set.

(ii) w_{2i} $(1 \le i \le 4)$ are the weights assigned to each type of component.

Table 4: The average frequency of the bigrams in UI-related and Non_UI-related callbacks (unit: times).

	Benign apps	Malicious apps
Non_UI	6.2	17.3
UI	11.6	9.4

Likewise, w_{2i} are also calculated by using a similar statistical approach. In the malicious samples and the benign ones, the average frequencies of the top 20 sensitive permissions in Table 2 are calculated for different component types, and the results are shown in Table 6. Similarly, $\varphi(c)$ are calculated by using formula (9) and then are normalized to get $\varphi'(c)$, as shown in Table 7. Finally, the values of $\varphi'(c)$ are assigned to w_{2i} $(1 \le i \le 4)$, respectively.

Table 5: φ (c) corresponding to UI and Non_UI callbacks.

	UI	Non_UI
$\phi(c)$	7.6	48.3
$\phi'(c)$	0.14	0.86

Table 6: The average frequencies of permissions at component granularity.

Component	Benign apps	Malware apps
Activity	4.142	6.147
Service	0.204	1.879
Receiver	0.283	2.954
Provider	0.042	0.290

4.2.3. Feature Function for Sensitive System Broadcasts. Since the context of a sensitive system broadcast is the application where the broadcast is registered, we assign a weight of 1 to this feature if it is registered in the application and 0 if unregistered.

In summary, the values of the feature vector in Figure 4 can be calculated using the following equations:

$$\begin{cases} a_i = \sum_{k=1}^{|\mathrm{UI}|} 0.14 + \sum_{k=1}^{|\mathrm{Non-UI}|} 0.86, \quad 1 \le i \le 300, \\ p_i = \sum_{k=1}^{|\mathrm{setActivity}|} 0.15 + \sum_{k=1}^{|\mathrm{setService}|} 0.30 + \sum_{k=1}^{|\mathrm{setReceiver}|} 0.52 + \sum_{k=1}^{|\mathrm{setProvider}|} 0.03, \quad 1 \le i \le 20, \\ c_i = \mu(\text{the corresponding broadcast is registered}), \quad 1 \le i \le 25, \end{cases} \tag{11}$$

where $\mu(\bullet)$ is the indicator function; it takes value 1 if $\bullet$ is true, otherwise takes value 0.

Example 2. We use a malware sample *Geinimi* to illustrate the values of a feature vector. *Geinimi* registers a broadcast event BOOT_COMPLETED in *manifest.xml* file to trigger its payload flexibly. Once the system is initialized, BOOT_COMPLETED is triggered and then the component AdServiceReceiver is launched. In the body of AdServiceReceiver. onReceiver(), URL.openConnection() and URLConnection.connect() are invoked to open a network connection. In the component AdCustomService, the malicious payload does something stealthy such as reading user contacts, sending text messages, and reading messages, which are privileged by the permissions READ_CONTACTS, SEND_SMS, and READ_SMS. Thus, the feature vector of *Geinimi* is shown as follows:

[..., 1.72,..., 0.14,..., 2.42,..., 0.86,..., 0.86,. . ., 0.86,. . ., 0.3,. . ., 0.3,..., 0.15,. . ., 0.15,. . ., 0.52,..., 1,. . ., 1,. . .], where the ellipses stands for the values of 0.

5. Implementation and Experiment

The implementation of our approach is based on Soot [19] and Sklearn library [20]. Soot is a Java bytecode optimization framework formerly developed by Sable Research Group of McGill University. It provides various kinds of analyses for Java programs, including IFDS/IDE dataflow analysis framework, Call graph construction and Point-to analysis. In this paper, we only use Soot to generate call graphs for applications and generate control flow graphs for every class methods. Sklearn library is a python-based third-party machine learning library; it integrates a variety of commonly used machine learning algorithms, which help us to train the classifiers. The implementation framework is shown in Figure 5.

Table 7: φ (c) and $\varphi'(c)$ corresponding to components.

Component	Activity	Service	Receiver	Provider
$\varphi(c)$	9	17	30	2
$\varphi'(c)$	0.15	0.30	0.52	0.03

5.1. Training Datasets. The malware samples are collected from the sample libraries provided by Drebin [5], Virus Share [21], and DroidBench of FlowDroid [12]. In order to make the samples more representative, the selected malware samples cover all malware families in each sample database. The benign samples come from Google Play, the official market of Android apps; they are tested by the computer butler and 360 security guard [22] in advance to ensure that they are benign applications. Table 8 shows the datasets adopted in this paper. In all of the samples, 75% of them are used as the training set and the remaining 25% are used as the test set. The processing time for each sample should not exceed 3 minutes.

5.2. Effectiveness of Graph Reduction Transformation. To demonstrate the necessity and effectiveness of the reduction transformation, we compare the complexities of original iCFGs and their reduced versions. The complexity of a graph is measured by the total number of nodes and edges of the graph.

In order to facilitate the processing, we select 300 APK files whose sizes are limited to less than 2 MB from the datasets. Figure 6 shows the comparison results between the complexities of the iCFGs and their corresponding API

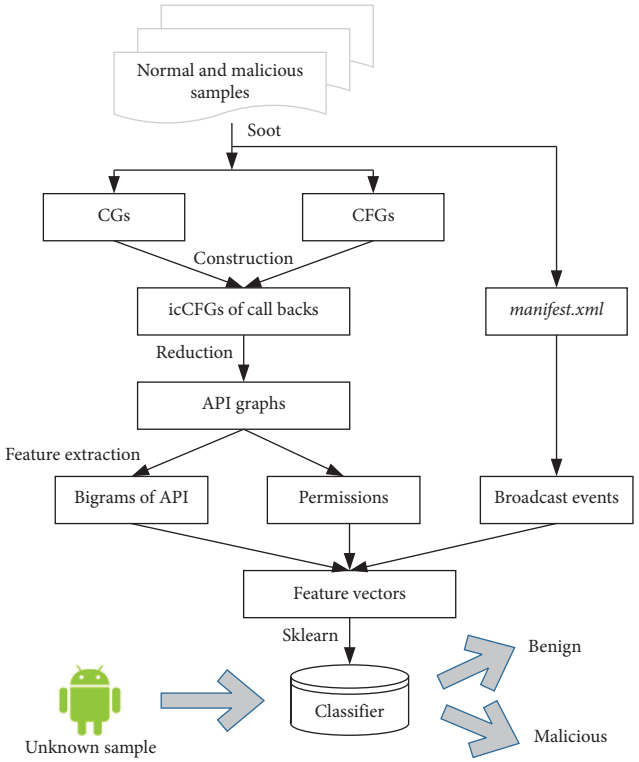

Figure 5: Implementation of our approach.

Table 8: The sample datasets.

APK type	Original sample libraries	Number of samples	Family
Malware	Drebin	1358	179
	Virus Share	1160	80
Benign	FlowDroid	142	Unclassified
	Google Play	2312	26

graphs. It shows that the average reduction rate (see formula (12)) is about 75.4%, implying that the number of nodes and edges in API graphs is significantly decreased after the reduction transformation.

$$P_{\text{reduction}} = 1 - \frac{(N - \text{API}) + (E - \text{API})}{(N - \text{ICFG}) + (E - \text{ICFG})}. \quad (12)$$

To further illustrate that the reduction transformation does improve the performance of feature generation in terms of time consumption, we carried out another comparative experiment, and the results are shown in Figure 7. It can be concluded that no matter how large the APK is, the time cost of the feature generation based on API graphs is lower than that based on iCFG. The average time cost (see formula (13)) is improved about 26.3% for the case of API graphs.

$$P_{\text{time-cost}} = \frac{(B - \text{cost}) - (A - \text{cost})}{B - \text{cost}} = 26.3\%. \quad (13)$$

5.3. Training a Detector. The detection of Android malware is treated as a classification problem, which can be solved

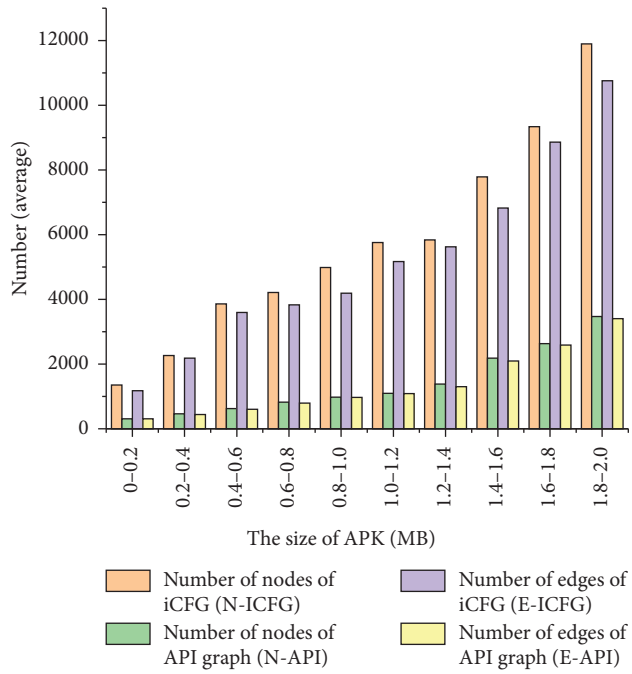

Figure 6: Comparison of the graph complexities.

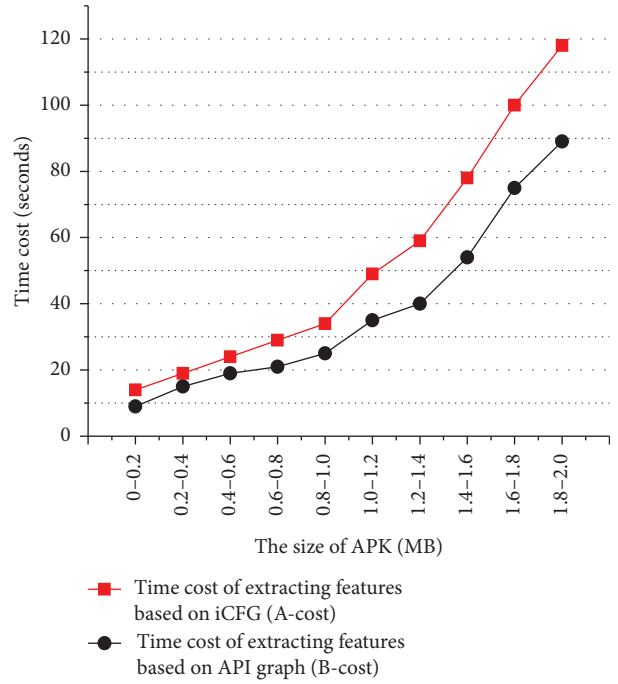

Figure 7: Comparison of time cost in feature generation.

effectively using machine learning techniques. The classification process is separated as two steps: *training* and *testing* the detector. All of 4972 samples are divided into two parts: 3732 for the training and 1240 for the testing. The training is a supervised learning process, which takes as the input the feature vectors labeled 1 or 0 (i.e., benign or malicious) and trains a learner (i.e., a detector) to detect malware; the testing process is used to evaluate the performance of the resulting detector.

Table 9: Detection results of different algorithms.

Classifier	NB (%)	KNN (%)	RF (%)	LR (%)	SVM (%)
Accuracy	89.2	93.8	95.4	93.1	88.4
FPR	12.5	7.9	5.6	9.1	13.1
Recall	90.7	95.2	96.5	95	89.7
Precision	89.2	93.2	95.1	92.3	88.7
F-Measure	89.9	94.2	95.8	93.6	89.2

As generally no absolutely optimal machine learning algorithms are available for a particular classification problem, we employ several classification algorithms to our feature set, including Naive Bayes (NB), k-nearest neighbor (KNN), random forest (RF), logistic regression (LR), and support vector machine (SVM). The detection performance of each algorithm is shown in Table 9.

From the results, we can conclude that RF classification algorithm has the best performance in the selected feature set, with a recall rate of 96.5%, an accuracy rate of 95.4%, and a false-positive rate of 5.6%. The performance of SVM algorithm is the worst.

5.4. Evaluation of the Selected Feature Set. In order to evaluate the performance of the selected features, we design and implement a number of comparison experiments. The overall experiments are organized as four groups: **Ex1**~**Ex4**, each of which evaluates the performance of one feature by leaving the other features fixed. The test cases of each group of experiments are shown in Table 10. For example, the experiment **Ex1** compares the *bigrams* of API calls with the *unigrams* of API call to observe the effectiveness of the bigrams with respect to the accuracy, recall, and precision rates. The experiments are implemented under the same machine learning classification algorithm RF, and the results are shown in Figure 8.

From Figure 8, we conclude that the features we choose present the best performance among the corresponding test cases. The reasons to contribute this achievement are analyzed as follows:

(1) The bigrams of API calls, as one of the raw features, can better reflect the behavioral characteristics of an application than individual API calls because a bigram of API calls contains a sequence of *two* consecutive API calls, which captures not only the APIs to be called but also the order how the APIs are invoked.

(2) When extracting the sensitive permissions from an application, we choose to acquire them from the source code instead of directly from the *manifest.xml* file, and thus the extracted permissions agree with those the application really uses. Using the permissions generated in such a way can achieve a better detection result.

(3) As the system broadcast events are frequently used by malicious payloads to trigger their behaviors, selecting them as one kind of feature will definitely improve the detection accuracy.

(4) Additionally, we combine the raw features with their contexts to form the new features, which hold richer semantic information than individual raw features, and thus using this feature set to classify an application can achieve a better detection performance.

5.5. Comparison with the Typical Works and the State-of-the-Art Tools. To justify the effectiveness of the proposed approach, we compare this approach with the typical works in recent 5 years. The results are shown in Table 11, from which we can conclude that the accuracy of our approach is better than most of other works, slightly less than DroidMat. Although DroidMat has the highest accuracy, its recall rate is lower than our approach. This result proves the effectiveness of the proposed approach.

Furthermore, we compare our approach with the state-of-the-art industry-scale detection tools. To this end, we apply our samples to VirusTotal, a free malware detection website that collects 69 malware detection tools. The comparison results are shown in Figure 9. It can be concluded that our detection performance is better than several mainstream tools in VirusTotal.

6. Related Works

Android malware detection has been extensively studied in recent years, and many methods and tools have been proposed to detect fast-growing Android malware. In this section, we overview the related works and give a brief classification for them.

6.1. Dynamic Analysis Approaches. Dynamic analysis approaches detect Android malware by monitoring an application's behavior at runtime. Taint analysis is the most typical approach of this kind; it is generally implemented in two ways: *platform-based* [9, 23] and *compiler-based* [24, 25]. The *platform-based* techniques need to customize or modify the underlying execution virtual machines (i.e., Dalvik VM or ART), which will automatically mark sensitive data, track the taint propagation, and provide feedback on the critical information to the analyzer to help detect potential misbehavior. Alternatively, the *compiler-based* techniques, instead of modifying the virtual machines, customize the *dex2oat*, the ahead-of-time (AOT) compiler adopted in Android 5.0 and above versions. The customized compiler can instrument taints and taint propagation rules into the original codes of an application when the application is being installed at the first time. SysDroid [7] proposes a dynamic analysis approach that uses Monkey Runner to mimic human interactions and extracts system API calls at runtime. It also proposes a new feature selection approach, known as SAIL, to discovering prominent system calls from applications and then uses machine learning techniques to detect potential malware samples. In addition, SysDroid concludes experimentally that the bigrams and trigrams of API calls present a better performance compared to unigram; this conclusion agrees with the results of our experiments (see Figure 8(a)).

Table 10: The design of the comparison experiments.

Experiments	APIs	Permissions	Broadcast events
Ex1	Bigrams of APIs Unigrams of API	Fixed	Fixed
Ex2	Bigrams with contexts Bigrams without contexts	Fixed	Fixed
Ex3	Fixed	Permissions with contexts Permissions without contexts Permissions in manifest.xml	Fixed
Ex4	Fixed	Fixed	With broadcast events Without broadcast events

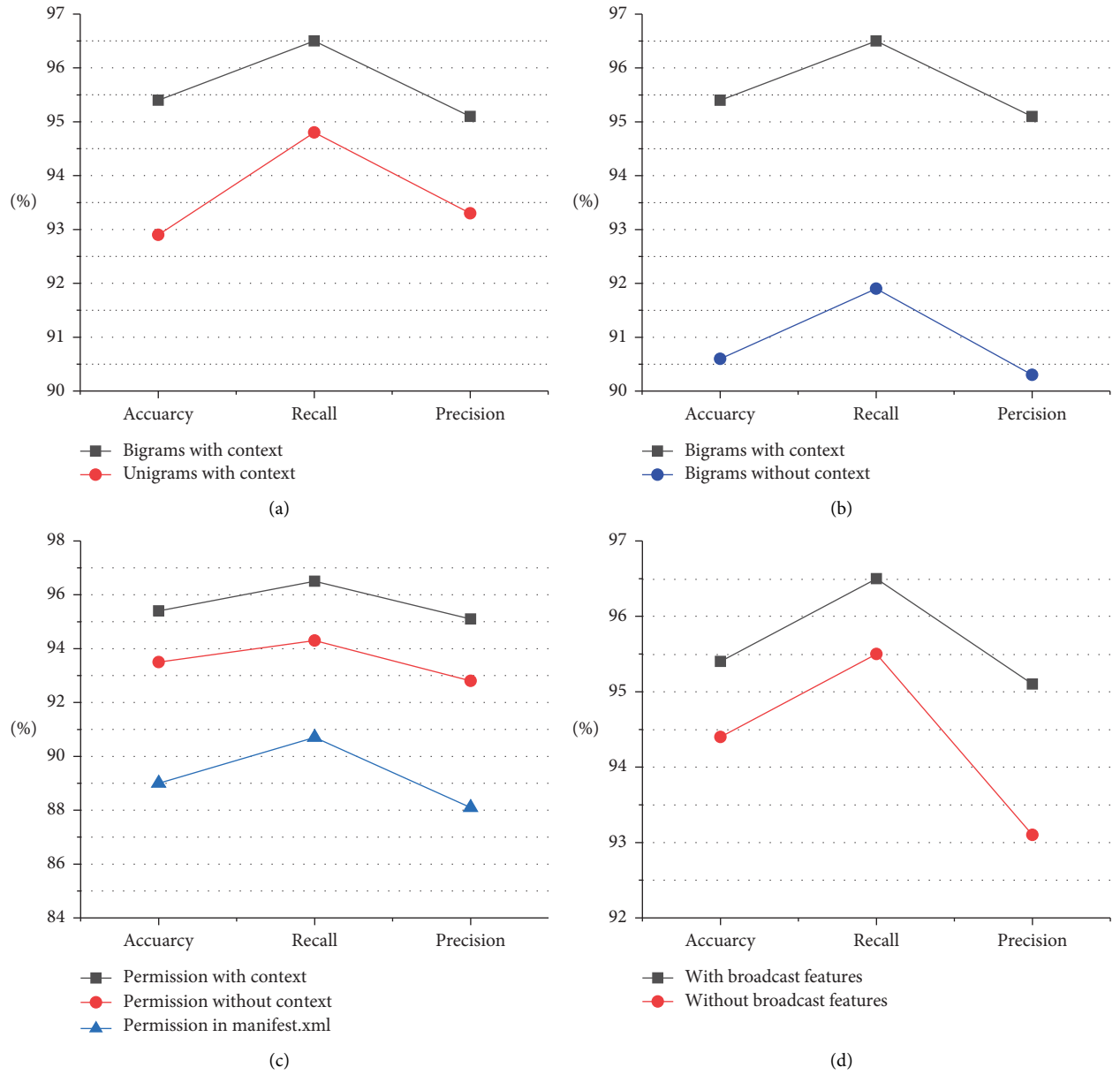

Figure 8: Evaluation of the selected features. (a) Ex1: bigrams vs. unigrams of APIs. (b) Ex2: bigrams with vs. without contexts. (c) Ex3: permissions in different cases. (d) Ex4: with vs. without broadcast events.

6.2. Static Analysis Approaches. Static analysis approaches are able to directly analyze the codes of programs without executing them yet. According to whether taking the details of programs into consideration, the static approaches are divided into two categories: *white box* [12, 26] approaches and *black box* approaches. White box approaches need to

Table 11: Comparison with the existing works.

Detection tool	Dataset		Feature set	Accuracy	Recall (%)
	Benign samples	Malicious samples			
DroidMat [10]	1500	238	Component, system call, API, etc.	97.8	87.4
Drebin [5]	123453	5560	Component, permission, intent filter, sensitive API, website address, etc.	93.9	95.9
Literature [4]	5000	1260	API and permission.	91.52	94.23
Literature [6]	3000	7449	API and permission.	94.6	90.7
Our approach	2600	2130	Sensitive permission, bigram of API calls, and sensitive system broadcast.	95.4	96.5

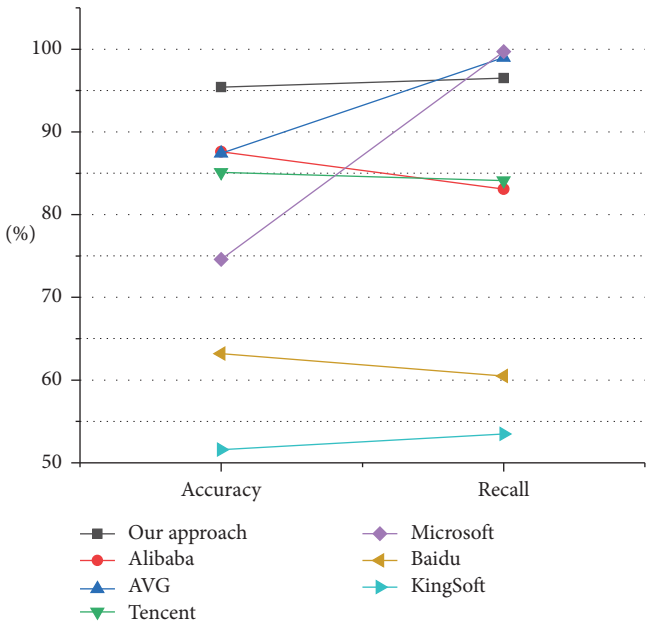

Figure 9: Comparison with existing mainstream tools.

fully investigate the internal structures or the semantics of programs and detect misbehavior by using program analysis techniques (such as dataflow analysis) or by discovering certain semantic inconsistencies in programs. These approaches usually suffer from the problems like higher complexity and poor scalability.

On the contrary, the black box methods do not analyze the internal structures and behaviors of programs but use statistical methods to detect malicious payloads. With the accumulation of a large number of malware samples and the rapid development of statistical machine learning methods, machine learning-based techniques have been widely used for the detection of malware [1–6, 11, 13, 14, 17, 27].

The work [8] is a typical work of using permissions and APIs as program features. It further differentiates permissions into standard permissions (defined by Android system) and nonstandard permissions (defined by developers). Permissions and APIs are extracted from the *manifest.xml* and *smali* files, respectively. Then, a feature selection technology (FST) is used to select the most prominent features in order to train the classifier. The feature set adopted by [8] is similar to ours, but it misses the contexts of features; therefore, some semantic information about the program would be lost. The work [28] extracts the second-step behavior features (SSBFs for short), i.e., what was triggered by the security-sensitive operations, to assist application analysis in differentiating between malicious and benign operations. The SSBFs include structural features (e.g., in-degree and out-degree of nodes in the iCFGs) and semantic features (such as the number of GUI behaviors and data-save behaviors). The work [28] also takes into account the dependency relations between the GUIs and API calls when considering program semantic characteristics; such dependency relations are similar to the contexts of the bigrams of API calls in our paper (see Section 4.2.1 for distinguishing UI and Non_UI callbacks). DaDiDroid [29] also performs feature extraction on the basis of Soot. It builds a directed weighted graph by modeling the calling relationship between APIs and then uses the graph structure as a feature to perform malware detection. Additionally, this method provides the flexibility for code obfuscation by abstracting specific API calls into API family names. Unlike our paper, DaDiDroid only focuses on the structures of programs but ignores other significant features such as permissions and system events.

6.3. Context-Based Approaches. Many detection methods take permissions, APIs, and their contexts as features [1, 3–6, 10, 11, 15, 30, 31]. AppContext [15], DroidSIFT [30], and DESCRIBEME [31] are typical works on context-based detection methods. Our work is partly motivated by the works AppContext and RepassDroid, but it improves them in the following two aspects. *First*, we combine the three kinds of raw features and their contexts as the features, while RepassDroid only takes security-sensitive APIs and permissions into consideration and merely considers the context for APIs; it thus loses the system events and the contexts of permissions which also hold semantic-rich information. *Second*, the generalized-sensitive APIs used in RepassDroid ignore the sequence of API calls, which is preserved in the bigrams of API calls, and thus in theory, our approach should achieve a better performance than RepassDroid.

7. Conclusion

In this paper, we propose a static approach to detecting Android malware, which focuses on the *context-based* feature selection and *graph-based* feature generation. We combine the three raw features with their contexts to form new features and utilize machine learning techniques to train the classifier. The evaluation results show that random forest is the optimal classification technique for our feature set, achieving 95.4% accuracy and 96.5% recall. At present, our approach can only give a binary classification for an application: malicious or benign, but cannot further distinguish the malware family of the application and reveal the impacts of malicious payload on the application's behaviors. These two weakness will be strengthened in the near future by improving the current work.

Data Availability

The malware samples are collected from the sample libraries provided by Drebin [5], Virus Share [21], and DroidBench of FlowDroid [12]; the benign applications are randomly collected from Google Play (https://play.google.com/store). All of these samples have been deposited in the Baidu cloud storage: https://pan.baidu.com/s/1UhKReGtEKSg7rLQDAUKZPQ. Please contact the authors for the password of access.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This study was supported by the Chinese NSFC Project (61702408), Science and Technology Plan Project of Shaanxi (2017JM6105), and Ministry of Education Collaborative Education Project "Mobile Operating System and Software Security Experimental Teaching Resources and Development" (2010918001).

References

[1] A. Mahindru and P. Singh, "Dynamic permissions based android malware detection using machine learning techniques," in *Proceedings of the 10th Innovations in Software Engineering Conference, ISEC*, pp. 202–210, Jaipur, India, February 2017.

[2] C. Li, M. W. Zhang, C. Y. Yang, and R. Sahita, "POSTER: Semi-supervised classification for dynamic android malware detection," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS*, pp. 2479–2481, Dallas, TX, USA, November 2017.

[3] B. Kang, S. Y. Yerima, and K. McLaughlin, "N-gram opcode analysis for android malware detection," *International Journal on Cyber Situational Awareness*, vol. 1, no. 1, pp. 231–255, 2016.

[4] M. Qiao, A. H. Sung, and Q. Liu, "Merging permission and API features for android malware detection," in *Proceedings of the IIAI International Congress on Advanced Applied Informatics*, pp. 566–571, Kumamoto, Japan, July 2016.

[5] D. Arp, M. Spreitzenbarth, and M. Hubner, "DREBIN: effective and explainable detection of android malware in your pocket," in *Proceedings of the Network And Distributed System Security Symposium*, pp. 1–15, San Diego, CA, USA, February 2014.

[6] A. A. Skovoroda and D. Y. Gamayunov, "Automated static analysis and classification of android malware using permission and API calls models," *Prikladnaya Diskretnaya Matematika*, no. 36, pp. 84–105, 2017.

[7] A. A. Ananya, P. Vinod, and M. Shojafar, "SysDroid: a dynamic ML-based android malware analyzer using system call traces," *Cluster Computing*, pp. 1–20, 2020.

[8] A. K. Singh, C. D. Jaidhar, and M. A. A. Kumara, "Experimental analysis of android malware detection based on combinations of permissions and API-calls," *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 4, pp. 1–10, 2019.

[9] W. Enck, P. Gilbert, B. G. Chun et al., "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones,," in *Proceedings of the 9th USENIX Symposium on Operating Systems*, pp. 309–407, Vancouver, Canada, October 2010.

[10] D. J. Wu, C. H. Mao, T. E. Wei et al., "DroidMat: android malware detection through manifest and API calls tracing," in *Proceedings of the 7th Asia Joint Conference on Information Security (JCIS)*, vol. 36, pp. 62–69, Tokyo, Japan, August 2012.

[11] N. Xie, F. Zeng, X. Qin et al., "RepassDroid: automatic detection of android malware based on essential permissions and semantic features of sensitive APIs," in *Proceedings of the International Symposium on Theoretical Aspects of Software Engineering, TASE*, pp. 52–59, Guangdong, China, August 2018.

[12] S. Arzt, S. Rasthofer, C. Fritz et al., "FlowDroid: precise context, flow, field, fbject-sensitive and 635 lifecycle-aware taint Analysis for android apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[13] R. Goyal, A. Spognardi, and N. Dragoni, "SafeDroid: a distributed malware detection service for android," in *Proceedings of IEEE International Conference on Service-Oriented Computing & Applications*, pp. 59–66, Macau, China, 2016.

[14] F. Wei, S. Roy, and X. Ou, "Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the ACM Conference on Computer And Communications Security*, pp. 1329–1341, Scottsdale AZ, USA, November 2014.

[15] Y. Wei, X. Xiao, B. Andow et al., "AppContext: differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the 37th International Conference on Software Engineering*, pp. 303–313, Florence, Italy, May 2015.

[16] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: a text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[17] M. Fan, J. Liu, X. Luo et al., "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.

[18] Z. Yajin and J. Xuxian, "Dissecting android malware: characterization and evolution," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, CA, USA, 2012.

[19] Soot, https://github.com/Sable/soot.

[20] SKlearn, https://sklearn.apachecn.org/.

[21] Virus Share, https://virusshare.com.

[22] 360 Security Guard, https://www.360.cn/.
[23] J. Li, Y. Ye, Y. Zhou, and J. Ma, "CodeTracker: a lightweight approach to track and protect authorization codes in sms messages," *IEEE Access*, vol. 6, no. 99, pp. 10107–10120, 2018.
[24] M. Backes, S. Bugiel, O. Schranz et al., "ARTist: the android runtime instrumentation and security toolkit," in *Proceedings of the IEEE European Symposium on Security and Privacy*, pp. 481–495, Pairs, France, June 2017.
[25] M. Sun, T. Wei, and J. C. Lui, "Taintart: a practical multi-level information-flow tracking system for android runtime," in *Proceedings of the Computer and Communications Security*, pp. 331–342, Vienna, Austria, October 2016.
[26] Y. Feng, S. Anand, and I. Dillig, "Apposcopy: semantics-based detection of android malware through static analysis," in *Proceedings of the Foundations of Software Engineering*, pp. 576–587, Hong Kong, China, November 2014.
[27] K. W. Y. Au, Y. F. Zhou, Z. Huang et al., "PScout: analyzing the Android permission specification," in *Proceedings of the ACM Conference on Computer and Communications Security, CCS*, pp. 217–228, Raleigh, NC, USA, October 2012.
[28] P. Li, J. Fu, C. Xu, B. Cheng, and H. Zhang, "Differentiating malicious and benign android app operations using second-step behavior features," *Chinese Journal of Electronics*, vol. 28, no. 5, pp. 944–952, 2019.
[29] M. Ikram, P. Beaume, and M. A. Kaafar, "DaDiDroid: an obfuscation resilient tool for detecting android malware via weighted directed call graph modelling," in *Proceedings of the 16th International Joint Conference on E-Business and Telecommunications*, pp. 211–219, Prague, Czech Republic, July 2019.
[30] M. Zhang, Y. Duan, H. Yin et al., "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 1105–1116, Scottsdale, AZ, USA, November 2014.
[31] M. Zhang, Y. Duan, Q. Feng et al., "Towards automatic generation of security-centric descriptions for android apps," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 518–529, Denver, CO, USA, October 2015.