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

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

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

\[
w(s, k): \text{ TF-IDF value of permission } s \text{ in category } k.
\]

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

\[
\text{totalNumber}(s) = \sum_{k=1,2} \text{num}(s, k). \tag{1}
\]

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

\[
w(s, k) = \text{per}(s, k) \times \log_{2}\frac{\text{allNumber}}{\text{totalNumber}(s)}. \tag{2}
\]

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.

### 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, \texttt{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

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

<table>
<thead>
<tr>
<th>Bigrams of API calls</th>
<th>Semantics of bigrams</th>
<th>Frequency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDeviceId();</td>
<td>Read the IMEI and IMSI values of the device</td>
<td>83.3</td>
</tr>
<tr>
<td>getSubscriberId()</td>
<td>Get IMSI value of sim card and send it via SMS</td>
<td>76.8</td>
</tr>
<tr>
<td>sendTextMessage();</td>
<td>Connect to the network and send data to the server</td>
<td>58.6</td>
</tr>
<tr>
<td>openConnection();</td>
<td>Get the sender’s address information and information content</td>
<td>56.7</td>
</tr>
<tr>
<td>getOutputStream();</td>
<td>Get the alarm service and take a broadcast as its response</td>
<td>43.6</td>
</tr>
<tr>
<td>getDisplayOriginatingAddress();</td>
<td>Intercept SMS address and SMS content</td>
<td>43.3</td>
</tr>
<tr>
<td>getSystemService(&quot;alarm&quot;);</td>
<td>Determine the network status</td>
<td>39.6</td>
</tr>
<tr>
<td>getBroadcast();</td>
<td>Get user’s geographic location</td>
<td>31.5</td>
</tr>
<tr>
<td>getOriginatingAddress();</td>
<td>Obtain IMEI ICCID values of the device</td>
<td>23.4</td>
</tr>
<tr>
<td>getMessageBody();</td>
<td>Query the database and get the query results</td>
<td>21.2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Sensitive permission ((s))</th>
<th>(w(s, k = 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND_SMS</td>
<td>0.243</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>0.239</td>
</tr>
<tr>
<td>READ_CALL_LOG</td>
<td>0.183</td>
</tr>
<tr>
<td>RECEIVE_SMS</td>
<td>0.182</td>
</tr>
<tr>
<td>READ_EXTERNAL_STORAGE</td>
<td>0.177</td>
</tr>
<tr>
<td>READ_CONTACTS</td>
<td>0.172</td>
</tr>
<tr>
<td>RECEIVE_BOOT_COMPLETED</td>
<td>0.171</td>
</tr>
<tr>
<td>CALL_PHONE</td>
<td>0.169</td>
</tr>
<tr>
<td>INSTALL_PACKAGE</td>
<td>0.167</td>
</tr>
<tr>
<td>ACCESS_WIFI_STATE</td>
<td>0.166</td>
</tr>
<tr>
<td>WRITE_EXTERNAL_STORAGE</td>
<td>0.165</td>
</tr>
<tr>
<td>MOUNT_UNMOUNT_FILESYSTEMS</td>
<td>0.165</td>
</tr>
<tr>
<td>READ_SMS</td>
<td>0.163</td>
</tr>
<tr>
<td>ACCESS_NETWORK_STATE</td>
<td>0.158</td>
</tr>
<tr>
<td>PROCESS_OUTCALLS</td>
<td>0.149</td>
</tr>
<tr>
<td>CHANGE_WIFI_STATE</td>
<td>0.148</td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>0.144</td>
</tr>
<tr>
<td>ACCESS_COARSE_LOCATION</td>
<td>0.140</td>
</tr>
<tr>
<td>INTERNET</td>
<td>0.140</td>
</tr>
<tr>
<td>WRITE_SETTING</td>
<td>0.091</td>
</tr>
</tbody>
</table>
EVENTS is a raw feature and application which listens to and for an event feature, is defined as a pair \((\text{features}), \text{a feature of an Android application}\). For a raw feature of bigram of API calls, its context is defined as a function context_bigram:

\[
\text{context_bigram}: \text{BIGRAMS} \rightarrow 2^{\text{CALLBACKS}}
\]

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

\[
\begin{align*}
\text{context_perm}: & \text{PERMS} \rightarrow 2^{\text{COMPS}} \\
\text{context_event}: & \text{EVENTS} \rightarrow \text{APPS}
\end{align*}
\]

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

\[\text{Definition 1}\] (contexts of the raw features). For a raw feature of bigram of API calls, its context is defined as a function context_bigram:

\[\text{context_bigram}: \text{BIGRAMS} \rightarrow 2^{\text{CALLBACKS}}\]

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

\[\text{Definition 2}\] (features). A feature of an Android application is defined as a pair \((f, c)\), where \(f \in \text{BIGRAMS} \cup \text{PERMS} \cup \text{EVENTS}\) is a raw feature and \(c = \text{context}_{\ast}(f)\) 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 OnReceive() and OnClick(), then its context is \{OnReceive(), OnClick\}, and the feature is written:

\[
\text{context}_{\ast}(f) = \{\text{OnReceive}, \text{OnDeviceId}, \text{getSubscriberId}\}.
\]

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 \textit{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 \textit{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 \textit{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 \textit{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

\[\text{Definition 3}\] (call graph). A call graph of an application \(a\) is defined as a labeled multigraph:

\[
\text{CG}_a = (N, S, E, r),
\]

where

\[\begin{align*}
(i) \ N & \text{ is a finite set of nodes, and each node is labeled a class method name in the set } \{m_1, m_2, \ldots, m_k\}. \\
& \text{The leaf nodes of } \text{CG}_a \text{ represent API calls.} \\
(ii) \ S & \text{ is a finite set of call-site labels, and each label represents a statement of method call.} \\
& \text{Each edge } e = (m_i, s_k, m_j) \in E \text{ denotes that class method } m_i \text{ calls class method } m_j \text{ at call site } s_k. \\
& \text{The unique entry node of } \text{CG}_a \text{ represents the root class method } m. \\
& \text{The call graph is actually a multigraph with multiple directed edges from one node to another or alternatively multiple labels on some edges.}
\end{align*}\]

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.

\[\text{Definition 4}\] (control flow graph). A CFG of a method \(m\) is defined as a directed simple graph

\[
\text{CFG}_m = (N, E, s, e),
\]

where
Steps of the feature generation:

(i) Generate a CG (call graph) for an application and then generate a collection of CFGs (control flow graphs), one for each class method inductively called by a callback.

(ii) Combine these CFGs to form an iCFG which represents the comprehensive behavior of the callback.

(iii) Apply the graph reduction transformation to the iCFG to generate a simplified version, from which the bigrams of API calls can be extracted using a graph algorithm.

(iv) Apply Pscout [4] map to the bigrams of API calls, mapping the sensitive APIs to the corresponding permissions.

(v) Pair each raw features with its context and finally generate the set of features.

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

\[
iCFG^*_0 = (N^*, E^*, s_0, e_0),
\]

where

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

(ii) $E^* = E^0 \cup E^1 \cup E^2$, in which $E^0 = \bigcup_{i \in \{0,1,2,\ldots,k\}} E^0_i$ is the collection of all ordinary control-flow edges; $E^1 = \bigcup_{i \in \{0,1,2,\ldots,k\}} E^1_i$ 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 $(n_1, n_2) \in E^1$ 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 a node $m$. A call edge $(n_1, s_i)$ and a return edge $(e_j, n_2)$ correspond to each other if $i = j$ and $(n_1, n_2) \in E^2$.

(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 $\text{Call}$, $\text{Return}$, $\text{Start}$, and $\text{Exit}$ 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:

\[
\text{Call} = \bigcup_{i \in \{0,1,2,\ldots,k\}} \text{Call}_i,
\]

\[
\text{Return} = \bigcup_{i \in \{0,1,2,\ldots,k\}} \text{Return}_i,
\]

\[
\text{Start} = \{s_0, s_1, \ldots, s_k\}
\]

\[
\text{Exit} = \{e_0, e_1, \ldots, e_k\}
\]

\[
\text{StructNodes} = \text{Call} \cup \text{Return} \cup \text{Start} \cup \text{Exit}
\]

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:

$$x_0 = u, x_1, x_2, \ldots, x_n = v,$$

where $(x_i, x_{i+1}) \in E$, for $i = 0, 1, \ldots, n-1$. In this case, we call $v$ 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 $v$.

**Definition 6 (reduction of iCFG).** Let $G$ be a finite set of iCFGs. The graph reduction transformation for iCFGs is a function $T: G \rightarrow G$ that transforms an iCFG $g = (N^*, E^*, s_0, e_0) \in G$ to a reduced iCFG $g' = (N^{'*}, E^{'*}, s'_0, e'_0)$:

$$T(g) = g',$$

where

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

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

(iii) $s'_0 = s_0$

(iv) $e'_0 = e_0$

For convenience, we refer to the reduced iCFGs as \texttt{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 $m_\rho$, $n_1$ and $n_2$ be two nodes in $N^*$, and \texttt{BIGRAM} be the set of bigrams of API calls appearing in $m_\rho$. Then, \texttt{BIGRAM} can be calculated using the following equation:

$$\texttt{BIGRAM} = \{l(n_0); l(n_1)| (n_0, n_1) \in E^0\} \cup \{l(n_0); l(n_1)| \exists p \in \text{Call, } q \in \text{Start } \bullet (n_1, p) E^0 \land (p, q) \in E^2 \land (q, n_2) \in E^2\} \cup \{l(n_0); l(n_1)| \exists p \in \text{Exit, } q \in \text{Return } \bullet (n_1, p) E^0 \land (p, q) \in E^2 \land (q, n_2) \in E^2\}.$$

**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 \texttt{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 $i (1 \leq i \leq 345)$ of the vector corresponds to a fixed raw feature, and the $i$th 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
Let $f_i$ be the $i$th raw feature in the vector; its context is $c_i = context \ast (f_i)$, 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, $\phi(c_i)$, can be calculated by applying a function $f$ to $(f_i, c_i)$: $\phi(c_i) = f(f_i, c_i)$.

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 $b \in \text{BIGRAMS}$ be a raw feature of bigrams; it constitutes a feature $(b, context\_bigram (b))$. The feature function on this feature is defined as

$$
f(b, context\_bigram (b)) = \begin{cases} 
\sum_{[\text{UI}]} w_{11} + \sum_{[\text{Non\_UI}]} w_{12}, & \text{if context\_bigram (b) \neq \emptyset}, \\
0, & \text{if context\_bigram (b) = \emptyset}, 
\end{cases}
$$

where

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

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

The values of $w_{11}$ 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 $\text{Non\_UI}$-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:

$$
\phi(c) = \frac{C_{m}^{2}(c)}{C_{n}^{2}(c)},
$$

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 $\phi(c)$ of each context. For convenience, $\phi(c)$ is normalized as $\phi'(c)$, which is assigned to the weight $w_{11}$, i.e., $w_{11}$ takes the value of $\phi'(\text{UI}) = 0.14$ and $w_{12}$ takes the value of $\phi'(\text{Non\_UI}) = 0.86$.

#### 4.2.2. Feature Function for Sensitive Permissions

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

$$
f(p, context\_perm (p)) = \begin{cases} 
\sum_{[\text{Activity}]} w_{21} + \sum_{[\text{Service}]} w_{22} + \sum_{[\text{Receiver}]} w_{23} + \sum_{[\text{Provider}]} w_{24}, & \text{if context\_perm (p) \neq \emptyset}, \\
0, & \text{if context\_perm (p) = \emptyset}, 
\end{cases}
$$
where

(i) \texttt{SetActivity}, \texttt{SetService}, \texttt{SetReceiver}, and \texttt{SetProvider} are mutually exclusive subsets of context_perm (p). For example, \texttt{SetActivity} is the set of activity components in context_perm (p). \abs{\texttt{Set}} is the cardinality of \texttt{Set}.

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

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, $\phi (c)$ are calculated by using formula (9) and then are normalized to get $\phi' (c)$, as shown in Table 7. Finally, the values of $\phi' (c)$ are assigned to $w_{2i}$ ($1 \leq i \leq 4$), respectively.
In this paper, we only use Soot to generate call graphs for frameworks, Call graph construction and Point-to analysis.

Java programs, including IFDS/IDE dataflow analysis framework formerly developed by Sable Research Group of McGill University. It provides various kind 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.
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.

\[ P_{\text{reduction}} = 1 - \frac{(N - \text{API}) + (E - \text{API})}{(N - \text{ICFG}) + (E - \text{ICFG})} \]  \hspace{1cm} (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.

\[ P_{\text{time-cost}} = \frac{(B - \text{cost}) - (A - \text{cost})}{B - \text{cost}} = 26.3\% \]  \hspace{1cm} (13)

5.3. Training a Detector. The detection of Android malware is treated as a classification problem, which can be solved 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.
5.4. Evaluation of the Selected Feature Set. In order to evaluate the performance of each algorithm, 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.

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

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 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 virtual machines (i.e., Dalvik VM or ART).
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>
<thead>
<tr>
<th>Experiments</th>
<th>APIs</th>
<th>Permissions</th>
<th>Broadcast events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex1</td>
<td>Bigrams of APIs</td>
<td>Fixed</td>
<td>Fixed</td>
</tr>
<tr>
<td></td>
<td>Unigrams of API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ex2</td>
<td>Bigrams with contexts</td>
<td>Fixed</td>
<td>Fixed</td>
</tr>
<tr>
<td></td>
<td>Bigrams without contexts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ex3</td>
<td>Fixed</td>
<td>Permissions with contexts</td>
<td>Fixed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Permissions without contexts</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Permissions in manifest.xml</td>
<td></td>
</tr>
<tr>
<td>Ex4</td>
<td>Fixed</td>
<td>Fixed</td>
<td>With broadcast events</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Without broadcast events</td>
</tr>
</tbody>
</table>

Table 10: The design of the comparison experiments.

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.
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 blackbox 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.

### Table 11: Comparison with the existing works.

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

![Figure 9: Comparison with existing mainstream tools.](image-url)

### 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 DESCRIEBME [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 weaknesses 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/1UhKReGtEK5g7rLQDAUKZPQ. 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