IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 10, 2020, accepted August 13, 2020, date of publication August 25, 2020, date of current version September 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3019282

Multifamily Classification of Android Malware With a Fuzzy Strategy to Resist Polymorphic Familial Variants

XIAOJIAN LIU, XI DU, QIAN LEI, AND KEHONG LIU
College of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an 710054, China

Corresponding author: Xiaojian Liu (780209965@qq.com)

This work was supported in part by the Science and Technology Plan Project of Shaanxi under Grant 2017JM6105, and in part by the Ministry of Education Collaborative Education Project Mobile Operating System and Software Security Experimental Teaching Resources and Development under Grant 2010918001.

ABSTRACT The Multifamily classification of Android malware aims to identify a malicious sample as one of the given malware families. This problem is believed to be much more significant than the *binary classification* (simply identify a sample as malicious or benign) because it is able to reveal the behaviour patterns of multiple malware families and bring deep insights into the working mechanism of malicious payload. The main challenges of the multifamily classification involve two aspects: recognizing the *behaviour patterns* of malware families as well as addressing the issues of *code obfuscation* and *polymorphic variants* that are commonly used by adversaries to evade rigorous detections. To address these challenges, in this article, we utilize the *regular expressions* of callbacks to describe the behaviour patterns of malware families, and propose a *two-step fuzzy processing strategy* to resist potential polymorphic familial variants. The alphabet of such regular expressions only consists of security-sensitive API calls, this enables the regular expressions to resist various kinds of code obfuscation and metamorphism. The proposed fuzzy strategy, applied to the regular expressions, comprises two steps: the first step transforms an original regular expression to such a fuzzy regular expression that possesses a broader meaning than the original one; the second step further relaxes precise plaintext match between two regular expressions to a fuzzy match by introducing the notion of *similarity* of regular expressions. Applying this strategy promotes the abstract level of a regular expression and enables the behaviour pattern specified by the regular expression to be more resilient to code obfuscation and polymorphic variants. Furthermore, selecting the fuzzy regular expressions as features, we use text mining techniques to train a multifamily 1-NN classifier over 3270 samples of 65 families. The experimental results show that our approach outperforms most of the state-of-the-art approaches and tools, confirming the effectiveness of our approach.

INDEX TERMS Android malware detection, malware family, regular expressions, fuzzy, text mining, k-NN classifier.

I. INTRODUCTION

The Android system, as one of the most popular mobile platform, still faces serious security challenges due to its open-source nature, imperfect design of the permission system, and the absence of a full certification to application publications. Malware or malicious payload exploits the vulnerabilities within the Android system to implement a variety of attacks, such as privilege escalation, remote control, financial charges and personal information stealing, severely threatening privacy protection, financial security, even social stability [1]. Therefore, it is very urgent to develop solutions to detecting and analysing the potential misbehaviour whenever installing and using an application.

The associate editor coordinating the review of this manuscript and approving it for publication was Feng Xia.

A. CHALLENGES

With a large number of malware samples being accumulated and publicly available, data mining and machine learning techniques provide an alternative perspective to detect and analyse malicious applications [2]–[10]. In this setting, the issue of malware detections can be treated as a problem of

classification, which can be tackled effectively by training an optimal classifier over massive malware samples.

Classification of Android malware can be *binary* or *multiclass* (or called *multifamily* in our setting). The binary classification is to simply distinguish an unkown application as *malicious* or *benign*; while the multifamily classification requires to further categorize a malicious sample into one of the multiple candidate families.

Obviously, the task of multifamily classification is tougher than that of the binary classification, because the former has to first recognize the behaviour of an unknown sample, then search for the best match of it against a set of family patterns. In spite of the presence of these difficulties, the multifamily classification is critically demanded for the obvious benefit to malware analysis, considering it is able to reveal the behaviour patterns of malware families and offer much more helpful insights into the working mechanism of malicious payload.

In general, any approach to the multifamily classification must properly address the following challenges only to achieve a better classification result:

1) HOW TO DESCRIBE THE BEHAVIOUR PATTERN OF AN ANDROID APPLICATION AND HOW TO MAKE SUCH A DESCRIPTION MORE RESILIENT TO CODE OBFUSCATION

To represent the behaviour patterns of malware families and samples, we shall first capture the essence of the program and present a concise but comprehensive description for the application to be analysed. Some works adopt a set of typical permissions [7], [8], (bigrams of) API calls [2], [9], [11], or system broadcasts [8], [9] to characterize the behaviours of Android applications; some works employ graphical forms, such as call graphs [12], control flow graphs [9], or other kinds of graphs [13]–[15], to represent the structure and behaviour of an application. Different descriptions will lead to different computational overheads and generate classfiers with different performance.

Furthermore, we demand the behaviour descriptions must be equipped with the capability to resist various kinds of code obfuscation and metamorphism, such as renaming user-defined functions, nested calls and control flow obfuscation, in order to combat against possible evasion from detections.

2) HOW TO EXTRACT COMMON BEHAVIOUR PATTERNS OF FAMILIES AND COPE WITH POSSIBLE POLYMORPHIC VARIANTS

A malware family is usually identified by a common behaviour pattern shared by all the samples within the family. In fact, the major work of multifamily classification is just to recognize the behaviour patterns of malware families. However, the common behaviour pattern of a family may suffer from somewhat variations. This situation may be raised either by some samples without exactly matching the common behaviour pattern, or by some samples that are deliberately designed by adversaries to evade rigorous detections. Therefore, we must properly deal with possible variants so as to effectively extract the common behaviour pattern for a family.

3) HOW TO IDENTIFY THE MOST DISTINGUISHABLE DISCRIMINANTS FOR FEATURE VECTORS AND HOW TO REDUCE THE DIMENSIONALITY OF FEATURE SPACE

To mitigate the problem of ''curse of dimensionality'' in machine learning process, we shall try to discriminate the most significant ones from a large number of candidate attributes to reduce the dimensionality of feature vectors and improve the performance of classification.

B. APPROACH

To address the above challenges, in this article, we propose an approach to the multifamily classification of malware by using text mining and machine learning techniques. We first overview the main blocks of our approach illustrated in Fig. 1. The implementation and the related datasets of our approach have been published in Github and Baidu Cloud Disk respectively: https://github.com/dasdasdf/Codes for code and https://pan.baidu.com/s/1Mia3mCcsL6D_BMGZ02jKKA with password ''x4re'' for datasets.

Our approach consists of two phases:

1) MODELLING PHASE

This phase aims to construct feature vectors for malware families, one feature vector for each family. The behaviour of an Android application is described by a set of behaviours of *callbacks*; the behaviour of a callback is specified by a *regular expression* that can be extracted from the reduced iCFG (Interprocedural Control Flow Graph) of the callback (Section III).

To cope with the possible variants of a family, we devise a *two-step fuzzy strategy* to manipulate the extracted regular expressions (Section IV). The first step maps a regular expression to a fuzzy one; the second step further relaxes an exact match to a fuzzy match of regular expressions by introducing the notions of *distance* and *similarity* for regular expressions.

Next, we calculate the *predominant discriminative behaviours* dFBeh($\mathcal{F}_i$) for each family, and select the most significant 3211 attributes to constitute the feature vectors for both families and unknown samples. Thereafter, we leverage the TF-IDF (Term Frequency-Inverse Document Frequency) indicators, widely used in text mining discipline, to quantify the behaviour characteristics of families, and finally formulate the feature vectors for overall 65 malware families (Section V-B).

2) CLASSIFICATION PHASE

Given an unknown (or test) application, we first extracts the regular expressions of callbacks of the application, then process them into fuzzy regular expressions and finally formulate its feature vector. To conduct the multifamily classification, we construct an 1-NN classifier that matches the application's

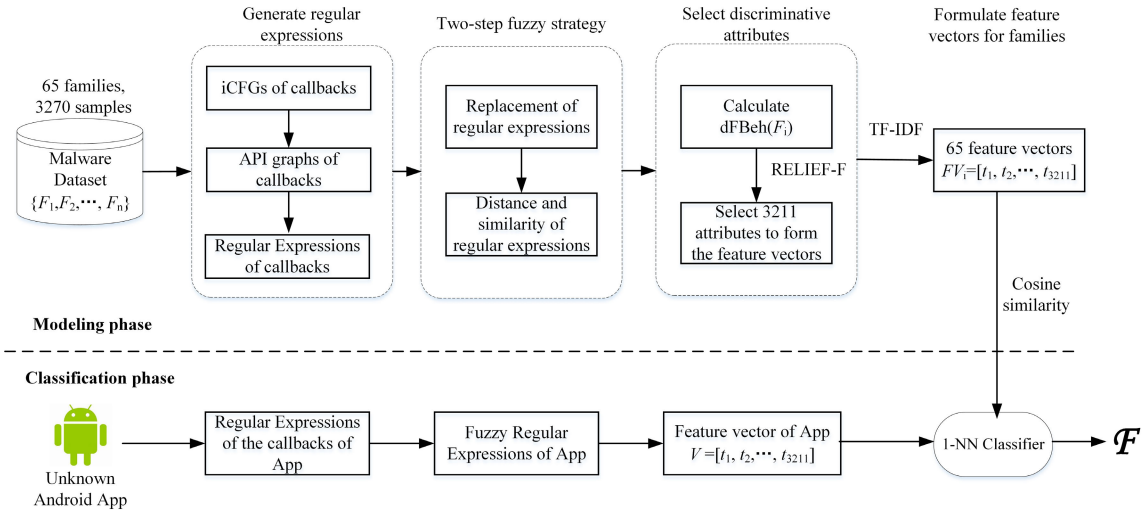


FIGURE 1. Overview of our approach.

feature vector with all familial feature vectors to find the nearest neighbor as its candidate family (Section V-D).

C. CONTRIBUTIONS

The main contributions of this work consist in the following four aspects, which respond the aforementioned challenges respectively:

1) We select callbacks rather than class methods as the basic units to describe an application's behaviour. As callbacks act as the entry points of an application's execution, their behaviours tend to offer much more semantics than class methods. The behaviour of a callback is further specified by a *regular expression*; the alphabet of such an expression only includes security-related API calls, enabling the regular expression to resist against code obfuscation.
2) We propose a *two-step fuzzy processing strategy* to tolerate the possible variations of a family. For a regular expression r, in the first step, we use a substitution operation h to map r to a new one r' (i.e., $h(r) = r'$), which covers a broader range of behaviour patterns than r; in the second step, the precise match is relaxed to a fuzzy match by introducing the concept of *similarity* between regular expressions. Through the both fuzzy processing steps, the abstraction level of a regular expression is promoted, and the behaviour pattern specified by the regular expression turns out to be more resilient to code obfuscation and polymorphic variants.
3) To achieve dimensionality reduction for feature vectors, we first calculate for each family $\mathcal{F}_i$ the predominant discriminative behaviour dFBeh($\mathcal{F}_i$) (a set of behaviours solely appear in family $\mathcal{F}_i$), then use RELIEF-F [16] technique to select 3000 additional attributes for feature vectors. In this way, the dimensionality of feature vectors is sharply reduced from 23568 to 3211, meanwhile a robust classification scheme is achieved as well.
4) With the constructed feature vectors, we trained a multifamily 1-NN classifier over 3270 samples of 65 families. The experimental results show that the average precision of our approach is about 97.8%, which outperforms against most of the state-of-the-art approaches and tools, evidencing the effectiveness of our approach.

The remainder of this article is organized as follows. Section II involves the behaviour descriptions for Android applications and families. Section III concentrates on extraction of regular expressions from iCFGs. Section IV introduces the two-step fuzzy strategy for the regular expressions, including the substitution rules and the notions of distance and similarity for the regular expressions. Section V focuses on the formulation of feature vectors for malware families, and construction of an 1-NN classifier for the multifamily classification. We implement our approach and report the experimental results in Section VI, and finally present related work in Section VII and conclusion in Section VIII.

II. DESCRIBE BEHAVIOURS OF MALWARE FAMILIES

A. BEHAVIOURS OF FAMILIES

Let $\mathcal{C} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ be a class of n malware families. A family $\mathcal{F}_i (0 \leq i \leq n)$ includes m_i sample applications in a given dataset, thus a family can be denoted

$$\mathcal{F}_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,m_i}\},$$

where $a_{i,j}$ is one of the sample applications in family $\mathcal{F}_i$.

The behaviour of application $a_{i,j}$ can be described by a set of *Callback Behaviours* (or *CB*s for short); each *CB* is a regular expression that specifies the behaviour of the callback. Formally,

Definition 1: (Behaviour of an application) The behaviour of an application $a_{i,j}$, denoted ABeh($a_{i,j}$), is defined as a set

of *CB*s

$$\mathrm{ABeh}(a_{i,j}) = \{CB_1, CB_2, \ldots, CB_p\},$$

where $CB_k \in N \times \mathcal{P}\Sigma^*$, $1 \leq k \leq p$; N is the set of all callbacks' names; p is the number of callbacks in $a_{i,j}$; Σ is the alphabet of regular expressions, namely a set of events to be observed; in our setting, the events are of the *generalized security-sensitive* API calls [7].

In what follows, we use the form (n, r) to denote the behaviour of a callback, where $n \in N$ is the callback's name, $r \in \mathcal{P}\Sigma^*$ stands for the regular expression associated with the callback.

The following considerations give the reasons why we choose callbacks (rather than class methods adopted in the work [17]) as the basic units to describe an application's behaviour:

1) The granularity of callbacks is more coarse-grained than that of class methods, leading to a smaller number of descriptions and lower dimensionality of feature vectors. This benefit will improve the efficiency of the classification.
2) Callbacks are usually triggered by various users and system events, and act as the entry points of execution of an Android application, therefore choosing them as the unit will allow us to fully capture the reactive behaviour of an application.

Example 1: The behaviour of the callback **Activity::onClick()** can be described as such a *CB*:

$$(\textbf{Activity::onClick()}, e_1; e_2; e_3; e_4^*)$$

where "**Activity::onClick**" is the name of the callback, **Activity** is the component type of **onClick()**; $e_1; e_2; e_3; e_4^*$ is a regular expression, each event e_i is concatenated with the sequential operator ";", the symbol "*" denotes a while loop. Here, e_1, e_2, e_3, and e_4 may represent API calls **getDeviceId()**, **getSubscriberId()**, **getDefault()** and **sendTextMessage()** respectively. Thus the regular expression reveals such a malicious scenario: once **onClick()** of an Activity component is invoked, the application will collect the information about "DeviceId" and "SubscriberId", and then send the collected information through calling the API **sendTextMessage()** one or more times.

Definition 2: (Behaviour of a malware family) Given a malware family $\mathcal{F}_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,m_i}\}$, the behaviour of $\mathcal{F}_i$, denoted $\mathrm{FBeh}(\mathcal{F}_i)$, is the union of behaviours of the applications in $\mathcal{F}_i$

$$\mathrm{FBeh}(\mathcal{F}_i) = \bigcup_{j=1}^{m_i} \mathrm{ABeh}(a_{i,j}).$$

Definition 3: (Common behaviour of a malware family) Given a malware family $\mathcal{F}_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,m_i}\}$, the common behaviour of $\mathcal{F}_i$, denoted $\mathrm{cFBeh}(\mathcal{F}_i)$, is the intersection of behaviours of the applications in $\mathcal{F}_i$

$$\mathrm{cFBeh}(\mathcal{F}_i) = \bigcap_{j=1}^{m_i} \mathrm{ABeh}(a_{i,j}).$$

In short, the set $\mathrm{cFBeh}(\mathcal{F}_i)$ contains those *CB*s that can be found in all samples of $\mathcal{F}_i$. Note that, $\mathrm{cFBeh}(\mathcal{F}_i)$ cannot be thought as the distinguishable behaviour pattern for family $\mathcal{F}_i$, because a common behaviour of a family may also appear in multiple families.

Definition 4: (Predominant discriminative behaviour) For a class $\mathcal{C} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ of n malware families, the *predominant discriminative behaviour* of $\mathcal{F}_i$, denoted $\mathrm{dFBeh}(\mathcal{F}_i)$, is a set of *CB*s that satisfies the following conditions:

1) $\mathrm{dFBeh}(\mathcal{F}_i) \subseteq \mathrm{cFBeh}(\mathcal{F}_i)$
2) $\forall j \bullet j \neq i \wedge \mathrm{dFBeh}(\mathcal{F}_i) \cap \mathrm{FBeh}(\mathcal{F}_j) = \emptyset$
3) (Maximum) If there is a set $\mathrm{dFBeh}'(\mathcal{F}_i)$ satisfying the conditions 1) and 2), then $\mathrm{dFBeh}'(\mathcal{F}_i) \subseteq \mathrm{dFBeh}(\mathcal{F}_i)$.

III. EXTRACT REGULAR EXPRESSIONS FROM CFGS

Each *CB* has two components: a callback's name and a regular expression that specifies the behaviour of the callback. In this section, we introduce how to extract the regular expression for a given callback; this process is illustrated by Fig. 2.

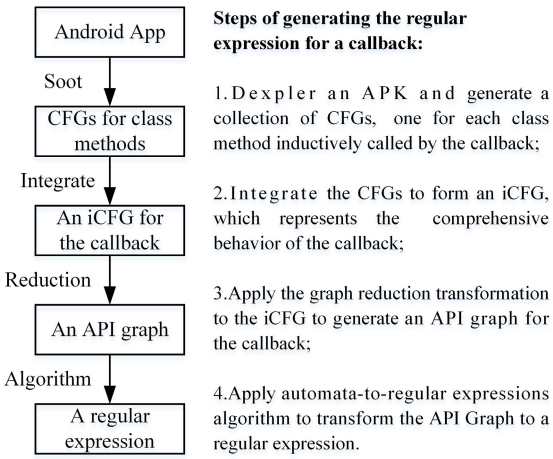


FIGURE 2. Generate the regular expression for a callback.

A graph transformational approach [9] is employed to extract the regular expressions. At first, an iCFG (Interprocedual Control Flow Graph) shall be constructed for a callback under the help of a certain static program analysis tool (we use Soot [18] in this article). An iCFG is a supergraph that integrates a collection of CFGs (Control Flow Graphs) together to describe the interprocedural calls of class methods. As we only concern with the security-sensitive API calls, in the next step, we reduce the generated iCFG to an API graph by removing irrelevant nodes and edges. Finally, we treat the generated API graph as a finite automaton, and use the automata-to-regular-expressions algorithm [19] to convert it to a regular expression.

In what follows, we only use a running example to show this process. The details of CFGs and the reduction transformation of iCFGs please refer to the authors' previous work [9].

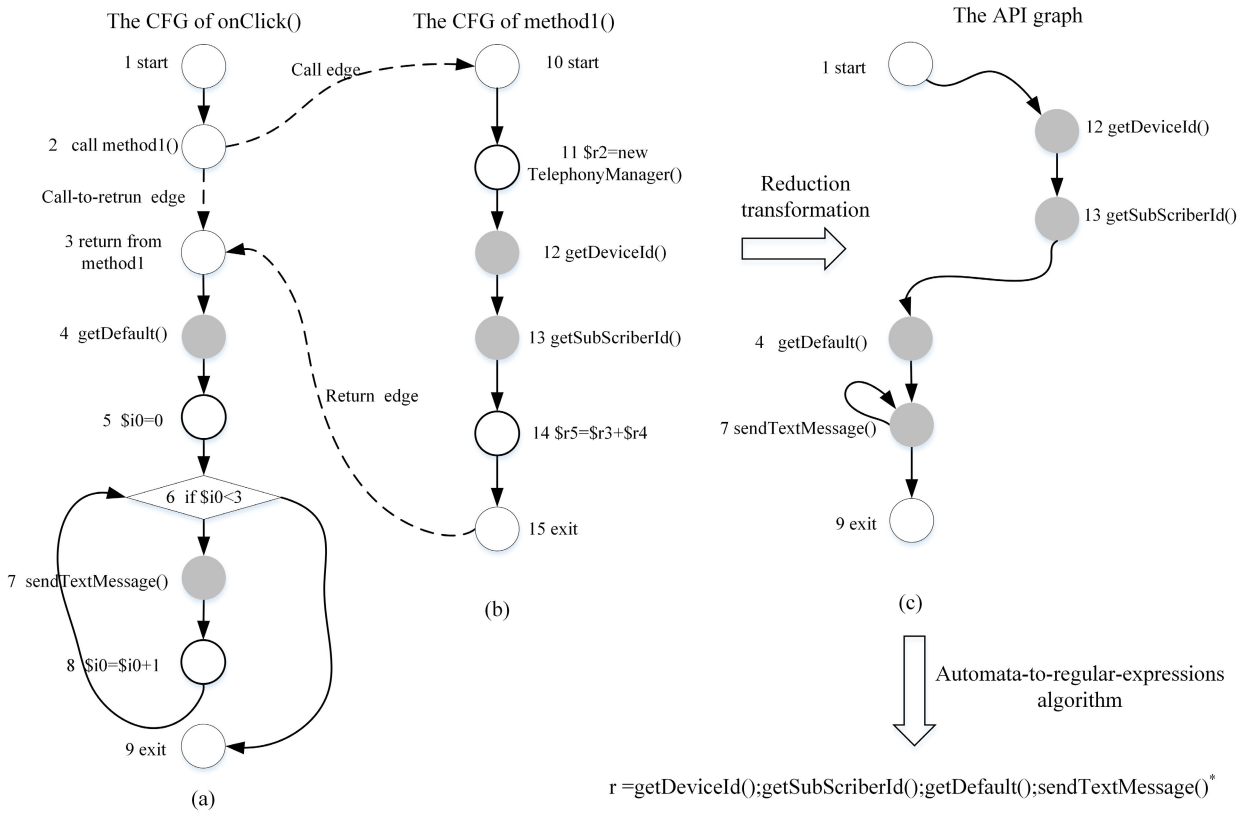


FIGURE 3. Extract a regular expression for a callback.

Example 2: Fig. 3 shows the process of extracting the regular expression from the iCFG of the callback onClick(). Subgraphs (a) and (b) are of the CFGs of onClick() and method1() respectively; they are connected through the edges "Call edge" and "Return edge" to form the iCFG of onClick(). Therein, the shaded nodes represent the security-sensitive API calls, other nodes represent either structural nodes (such as "start" and "exit" nodes) or the nodes that are labelled with security-irrelevant API calls or statements. The original iCFG is reduced to the API graph that only preserves the "start" and "exit" nodes of onClick() and the security-sensitive API call nodes, as illustrated by Subgraph (c). Finally, the regular expression is extracted from the API graph by using the automata-to-regular-expressions algorithm.

IV. THE TWO-STEP FUZZY PROCESSING STRATEGY FOR THE REGULAR EXPRESSIONS

A malware family usually derives numerous polymorphic variants, namely similar but not exactly the same behaviour patterns. If we would simply make use of the precise match to compare two regular expressions and calculate cFBeh($\mathcal{F}_i$) and dFBeh($\mathcal{F}_i$), then it might be too stringent to find out a meaningful behaviour pattern for a family.

To tackle this problem, we propose a two-step fuzzy processing strategy to deal with the possible variants of a family. The first step performs a substitution operation on an original regular expression to produce a fuzzy regular expression that covers a broader range of behaviour patterns than the original one; the second step further relaxes the exact match to a fuzzy match over the produced fuzzy regular expressions in an effort to tolerate somewhat variations to the common behaviour pattern of a family.

A. THE FUZZY PROCESSING OF REGULAR EXPRESSIONS

Essentially, the fuzzy processing is a substitution operation on a regular expression; that is, every event (i.e., API call) in a regular expression is replaced by a more coarse-grained event according to some given rules. This processing widens the meaning of each event and enables the regular expression to cover a broader range of behaviour patterns.

In what follows, we first present the substitution rules for an event, then define the substitution operation on a regular expression.

Definition 5: (Substitution rules) Let e be an event in a regular expression r. The substitution of e is an operation h that maps e to a new symbol (or a set of symbols) according to the rules shown in Table 1.

A rule is of a form

$$LHS \Longrightarrow RHS,$$

where LHS is the condition part of the rule, and RHS is the rule's conclusion part. The symbol $\mapsto$ means a substitution

TABLE 1. Substitution rules for an event.

$\mathbf{R_1}$	e is a *permission-guarded API call* $\Longrightarrow P \mapsto e$
$\mathbf{R_2}$	e is a *source* (*sink*) API $\Longrightarrow$ *source* (*sink*)-name $\mapsto e$
$\mathbf{R_3}$	e is an API related to *dynamic loading* $\Longrightarrow$ " DYNLOAD" $\mapsto e$
$\mathbf{R_4}$	e is a *security-sensitive* API $\Longrightarrow$ " SENAPI" $\mapsto e$
$\mathbf{R_5}$	The rules $\mathbf{R_i}(1 \leq i \leq 3)$ prioritize the rules $\mathbf{R_{i+1}}$.

relation, therefore $e' \mapsto e$ states that "replace event e with a new event e'".

The rule $\mathbf{R}_1$ means that "If e is a permission-guarded API call, then it is replaced with the guarded permissions P of e." The intuition behind this rule is that the guarded permissions have a more abstract and essential intention than the corresponding API call, and can cover multiple similar API calls with the same intention. For example, getSubscriberId() is guarded by the permission READ_PHONE_STATE; this permission actually means such an intention that the program intends to do a "read" operation (may be getSubscriberId() or getDeviceId()) on the resource "phone". Replacing this API with the permission READ_PHONE_STATE can capture the essence of the API call, and enables the guarded permission to cover a number of similar API calls such as getSubscriberId(), getDeviceId(), or some other APIs with the same intention.

The rule $\mathbf{R}_2$ means that "If e is a source (or sink) API, then we use the name of the source (or sink) to replace e." The source (or sink) API means the API identified in SuSi project [20]. If such an API appears in a regular expression, then it is replaced with the source- or sink-resource name of the API. For example, getMacAddress() is a source API, it is replaced with the symbol "SRC_Mac" where "Mac" is the source-resource name of this API; insert() is a sink API, it will be replaced by the symbol "SINK_CP" where "CP" means "ContentProvider", the sink-resource name of the API.

The rule $\mathbf{R}_3$ means that "If e is a dynamic loading API, then it is replaced with the symbol " DYNLOAD". For a dynamic-loading API, it is readily replaced with a constant symbol " DYNLOAD", meaning that this API is about the dynamic loading function, but we don't care about what it really is. Similar way is also applied to $\mathbf{R_4}$—for an API other than the APIs in $\mathbf{R_1} \sim \mathbf{R_3}$, it is replaced with a constant symbol " SENAPI".

If there exists a conflict in the application of the rules $\mathbf{R_1} \sim \mathbf{R_4}$, then the conflict is resolved by the priorities of the rules, which are assigned via $\mathbf{R_5}$.

The substitution operation for some typical events is illustrated in Table 2.

The domain of the substitution operation h can be generalized from events to regular expressions in a straightforward fashion. For example, given a regular expression

$$r = e_1; (e_2 \mid e_3); (e_4; e_5)^*,$$

r can be transformed to a new regular expression $h(r)$ by h

$$h(r) = h(e_1); (h(e_2) \mid h(e_3)); (h(e_4); h(e_5))^*.$$

TABLE 2. The substitution operation for some typical events.

e	$h(e)$
getSubscriberId	{READ_PHONE_STATE}
markAsContacted	{WRITE_CONTACTS, READ_SOCIAL_STREAM}
getConnectionInfo	{ACCESS_WIFI_STATE, DUMP}
setTestProviderStatus	{ACCESS_MOCK_LOCATION}
getLastOutgoingCall	{ADD_VOICEMAIL, READ_CALL_LOG, DUMP}
disconnect	{BLUETOOTH_ADMIN, DUMP}
addCompletedDownload	{WRITE_EXTERNAL_STORAGE, VIBRATE, INTERNET}
openConnection	{INTERNET}
getMacAddress	SRC_Mac
insert	SINK_CP
loadClass	DYNLOAD
generateSecret	SENAPI

Note that, through the substitution operation h, a callback behaviour $CB = (n, r)$ is transformed to a fuzzy callback behaviour

$$CB' = (n, h(r)).$$

But for convenience, in what follows, we still use CB to denote the fuzzy version of the callback behaviour; all the regular expressions mentioned thereafter, if no confusion, will refer to the fuzzy ones that have been processed through the substitution operation.

B. THE EDIT DISTANCE OF REGULAR EXPRESSIONS

The second step of the fuzzy processing strategy is concerned with the *fuzzy* match between the regular expressions. The fuzzy match is made by comparing the *similarity* between two regular expressions. The similarity is measured by the *distance* of the regular expressions.

The edit-distance of two regular expressions can be defined as follows.

Definition 6: (Edit-distance of regular expressions) [21] The *edit-distance* of two regular expressions r_1 and r_2, denoted $d(r_1, r_2)$, is defined as:

$$d(r_1, r_2) = \inf[d(w_1, w_2) : w_1 \in \mathbf{L}(r_1), w_2 \in \mathbf{L}(r_2)]$$

where $\mathbf{L}(r)$ is the language (i.e., the set of strings) of the regular expression r; w_1, w_2 are strings of events, $d(w_1, w_2)$ is the edit-distance of w_1 and w_2.

The distance $d(r_1, r_2)$ can be computed by the aid of two algorithms: *composition* of weighted automata [22] and *single-source shortest-paths* of graphs [21]. The skeleton of the algorithm is presented in Algorithm 1.

At first, the regular expressions r_1 and r_2 are converted to the corresponding automata A_1 and A_2 using traditional *regular-expression-to-automaton* algorithm [19] (lines 1-2); then compute the composition of A_1 and A_2 to obtain a new automaton A (line 3); finally, the distance d is computed using classical shortest-distance algorithm (line 4) such as Bellman-Ford dynamic programming algorithm.

Let's estimate the time complexity of the algorithm. Conversion of a regular expression to an ordinary NFA (without ϵ transition) takes $O(n^3)$ time on the regular expression of

Algorithm 1 Compute the Distance of the Regular Expressions

Input:
Regular expressions r_1 and r_2
Output:
$d(r_1, r_2)$: distance of r_1 and r_2
A_1 = regular-expression-to-automaton(r_1);
A_2 = regular-expression-to-automaton(r_2);
A = composition(A_1, A_2);
d = single-source-shortest-distance(A, q_0, q_f);

length n. Computing the composition of two automata takes $O(|A_1||A_2|)$ time, where $|A_i| = |Q_i| + |E_i|$, Q_i and E_i are of the sets of states and transitions of A_i respectively. The Bellman-Ford dynamic programming algorithm will take $O(|Q|^3)$ time, where Q is the set of states of A. In conclusion, the total time complexity of the algorithm is

$$O(n^3) + O((|Q_m| + |E_m|)^2) + O(|Q_m|^6),$$

where $|Q_m| = \max(|Q_1|, |Q_2|)$, $|E_m| = \max(|E_1|, |E_2|)$.

C. THE SIMILARITY OF REGULAR EXPRESSIONS

The similarity of regular expressions can be measured in terms of the distance defined in the previous section.

Definition 7: (Similarity of the regular expressions) The similarity of two regular expressions r_1 and r_2 is defined as

$$sim(r_1, r_2) = 1 - \frac{d(r_1, r_2)}{d(r_1, \emptyset) + d(\emptyset, r_2)},$$

where $\emptyset$ is the empty regular expression, $d(r_1, \emptyset) + d(\emptyset, r_2)$ is then equal to the maximum edit cost to transform r_1 into r_2.

With the notion of similarity, we can relax the exact plaintext match between the regular expressions to a fuzzy match; that is, two regular expressions r_1 and r_2 are considered to be *the same* if their similarity is greater than an acceptable threshold, i.e., $\theta \leq sim(r_1, r_2)$. The notion "*the same*" can be generalized to the callback behaviours in a straightforward way.

Definition 8: (The same callback behaviours) Two callback behaviours (n_1, r_1) and (n_2, r_2) can be considered to be the same if and only if $n_1 = n_2$ and $\theta \leq sim(r_1, r_2)$, where θ is a given threshold ranging from 0 to 1.

In this article, we assume $\theta = 90\%$. If two callbacks have the identical names, and the similarity of their regular expressions is greater than 90%, then the both can be seen the same.

D. CALCULATE THE COMMON BEHAVIOURS OF FAMILIES BASED-ON THE SIMILARITY

We use the aforementioned notion of similarity to improve the calculations of common behaviours cFBeh($\mathcal{F}_i$) and predominant discriminative behaviours dFBeh($\mathcal{F}_i$) of malware families $\mathcal{F}_i$ (see Section II-A).

To investigate the necessity and effectiveness of the similarity-based approach, we first conduct an experiment to compare this approach with the exact plaintext match approach. The experimental results are illustrated in Fig. 4.

The experiment counts the number of callback behaviours in the sets cFBeh($\mathcal{F}_i$) and dFBeh($\mathcal{F}_i$). The sets cFBeh($\mathcal{F}_i$) and dFBeh($\mathcal{F}_i$) have two versions respectively: one is calculated by using the similarity-based approach ($\theta = 90\%$), the other by the plaintext match approach. To distinguish both versions, we use cFBeh($\mathcal{F}_i, \theta$) and dFBeh($\mathcal{F}_i, \theta$) to denote the similarity-based versions; cFBeh($\mathcal{F}_i$) and dFBeh($\mathcal{F}_i$) just for the exact match ones. Note that, dFBeh($\mathcal{F}_i, \theta$) only derive from cFBeh($\mathcal{F}_i, \theta$).

The results show that the similarity-based approach can extract more callback behaviours than the plaintext match one. In particular, in some families, such as **Boxer**, **FakeInstaller**, and **Zitmo**, etc., the similarity-based approach can extract the common behaviours that cannot be discovered by the plaintext match approach.

Moreover, the results show that among all malware families, only 8 families have empty dFBeh($\mathcal{F}_i, \theta$), and the remaining 57 families have non-empty dFBeh($\mathcal{F}_i, \theta$). From this observation, it follows that, in most cases, the discriminative common behaviours dFBeh($\mathcal{F}_i, \theta$) can serve as a better candidate feature to classify an unknown sample.

V. MULTIFAMILY CLASSIFICATION BASED-ON TEXT MINING

We treat the multifamily classification as a problem of text classification—the callback behaviours of an application can be viewed as "words"; the application can be viewed as a "text", namely a collection of the "words"; and apparently, a malware family can be regarded as a "class" of "texts". Therefore, classifying an unknown application to a malware family amounts to the classification of a "text" into one of the text "classes". By using the traditional text mining techniques, we can construct a text classifier to resolve the problem of familial classification. The main challenges of this solution consist in the *feature selection* and *classifier construction*, which will be addressed in this section.

A. FEATURE SELECTION WITH THE RELIEF-F APPROACH

In principle, the sets dFBeh($\mathcal{F}_i, \theta$) should be directly selected as the features to distinguish malware families for an unknown sample, but these features may suffer from somewhat fragility in nature.

Consider this case where a new malicious sample is classified into a certain family, but it was found no common callback behaviours shared with all the other samples in the family. This may be because the sample was either misclassified, or deliberately designed to evade rigorous detections. In this case, both cFBeh($\mathcal{F}_i, \theta$) and dFBeh($\mathcal{F}_i, \theta$) turn out to be empty; that is, no discriminants can be found to identify the family.

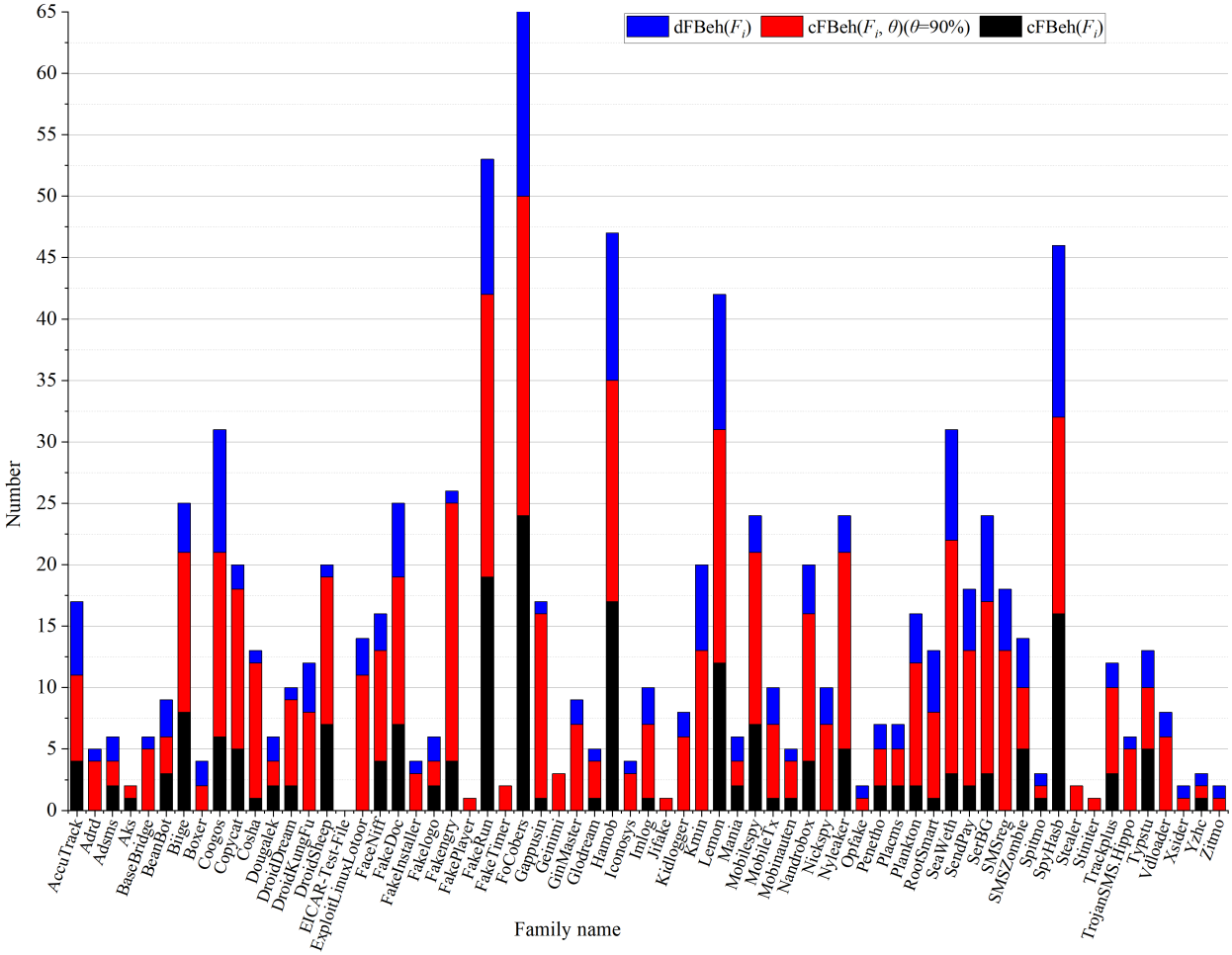


FIGURE 4. Distributions of cFBeh($\mathcal{F}_i$), dFBeh($\mathcal{F}_i$), and dFBeh($\mathcal{F}_i, \theta$)($\theta = 90\%$) over the families.

To tolerate such a fragility and select a more robust feature set for the classification, we construct the feature vectors in this way:

- First include all callback behaviours in dFBeh($\mathcal{F}_i, \theta$) of 57 families (notice only 57 families present nonempty dFBeh in total 65 families) as the attributes to the feature vectors,
- Then search for k additional callback behaviours as extra attributes to join in the feature vectors.

Therefore, the feature vectors shall be formulated as the form

$$Vector = [\overrightarrow{\text{dFBeh}(\mathcal{F}_1, \theta)}, \ldots, \overrightarrow{\text{dFBeh}(\mathcal{F}_{57}, \theta)}, f_1, \ldots, f_k],$$

where $f_i (1 \leq i \leq k)$ are of the additional k attributes.

To this end, we leverage the RELIEF-F approach [16] to select k candidate callback behaviours as the additional attributes $f_i (1 \leq i \leq k)$.

Following the RELIEF-F approach, we select the most contributive extra attributes of the feature vectors according to these steps:

1) Assign each of 23357(= 23568−211) (note: 65 dFBeh($\mathcal{F}_i, \theta$) have included 211 callback behaviours in total) callback behaviours a serial number ranging from 1 to 23357, and represent every malicious sample as a vector $\boldsymbol{x}$ containing 23357 attributes

$$\boldsymbol{x} = [m_1, m_2, \ldots, m_{23357}, \mathcal{F}],$$

where $\mathcal{F}$ is the family label of $\boldsymbol{x}$, the values of $m_i (1 \leq i \leq 23357)$ may be 0 or 1 defined as follows

$$m_i = \begin{cases} 1 & \text{the sample has the } i\text{-th callback behaviour} \\ 0 & \text{otherwise} \end{cases}$$

2) Randomly select 20 samples from every family to collect 1300(= 20 × 65) samples in total. Represent each sample as a vector and finally form a dataset $D = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{1300}\}$ of 1300 sample vectors.
3) Apply the RELIEF-F approach to the dataset D to select the most significant k attributes from 23357 ones.

Using this approach, we select 3000 (i.e., $k = 3000$) callback behaviours as the additional attributes to the feature vectors. The following Fig. 5 illustrates the selected top 30 attributes and their weights.

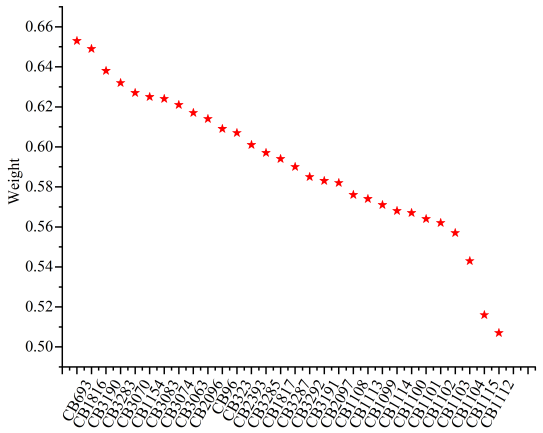


FIGURE 5. Top 30 callback behaviours and their weights calculated using the RELIEF-F approach.

B. CONSTRUCT FEATURE VECTORS FOR MALWARE FAMILIES

For each malware family, we shall construct a familial feature vector to model its predominant characteristics. A familial feature vector is a vector with overall 3211 attributes

$$[\overrightarrow{\text{dFBeh}(\mathcal{F}_1, \theta)}, \ldots, \overrightarrow{\text{dFBeh}(\mathcal{F}_{57}, \theta)}, f_1, \ldots, f_{3000}],$$

in which $\text{dFBeh}(\mathcal{F}_i, \theta)(1 \leq i \leq 57)$ contain 211 attributes together with extra 3000 the most contributive attributes.

Given a training set, we shall calculate the weight of each attribute in a familial feature vector. We borrow the indicator *TF-IDF* (*Term Frequency-Inverse Document Frequency*) [23], widely used in text mining discipline, as the weight of an attribute. For the feature vector FV_i of the family $\mathcal{F}_i$

$$FV_i = [t_{i,1}, t_{i,2}, \ldots, t_{i,3211}],$$

the weights $t_{i,j}(1 \leq j \leq 3211)$ are calculated by

$$t_{i,j} = \text{TF}(CB_j, \mathcal{F}_i) \times \text{IDF}(CB_j, \mathcal{C}),$$

where $\text{TF}(\bullet)$ is the term frequency of the callback behaviour CB_j over the family $\mathcal{F}_i$; $\text{IDF}(\bullet)$ is the inverse document frequency of CB_j over family class $\mathcal{C}$. Both terms are calculated as follows.

Definition 9: (Term frequency) $\text{TF}(CB_j, \mathcal{F}_i)$ represents the frequency of a callback behaviour CB_j over the family $\mathcal{F}_i$, which can be calculated by the following formula

$$\text{TF}(CB_j, \mathcal{F}_i) = \frac{\sum_{a \in \mathcal{F}_i} \text{freq}(CB_j, a)}{|\text{FBeh}(\mathcal{F}_i)|},$$

where $\text{freq}(CB_j, a)$ is the number of times CB_j appears in the application a.

Definition 10: (Inverse document frequency) $\text{IDF}(CB_j, \mathcal{C})$ represents the inverse document frequency of a callback behaviour CB_j over family class $\mathcal{C}$, which can be calculated by the formula

$$\text{IDF}(CB_j, \mathcal{C}) = \log \frac{|\mathcal{C}|}{1 + |\{\mathcal{F}_k \in \mathcal{C} | CB_j \in \text{FBeh}(\mathcal{F}_k)\}|}.$$

When a certain callback behaviour does not appear in any family, it will cause the denominator to be zero; therefore we add 1 to the denominator to cope with this situation.

Example 3: We give an intuitive example to illustrate the calculation of TF-IDF for a callback behaviour. Suppose a family class $\mathcal{C}$ includes 100 malware families, i.e., $\mathcal{C} = \{\mathcal{F}_1, \ldots, \mathcal{F}_{100}\}$. The distribution of the callback behaviour CB_j over the families is shown in Table 3.

TABLE 3. TF-IDF of CB_j in malware family class $\mathcal{C}$.

	$\mathcal{F}_1$			$\mathcal{F}_2$			...	$\mathcal{F}_{100}$
samples	a_1	a_2	a_3	a_4	a_5	a_6	...	a_n
CB_j in	✓	✓	✓	✗	✗	✓	✗	✗
TF	1			1/3			0	0
IDF	$\log \frac{100}{1+2} = 1.523$							
TF-IDF	1.523			0.508			0	0

Table 3 shows that CB_j has the highest weight in family $\mathcal{F}_1$, because all applications in $\mathcal{F}_1$ contain CB_j; for those families that do not contain CB_j, the corresponding weights are all 0's.

C. THE ALGORITHM TO CALCULATE FAMILIAL FEATURE VECTORS

In the overall 3270 malware samples of 65 families, we randomly select a total of 1300 samples (20 samples for each family) to constitute the training set, and the remaining data set as test samples.

The algorithm for constructing the familial feature vectors is presented as Algorithm 2, which mainly consists of the three steps:

1) The first step (lines 1–7) calculates the sets of callback behaviours for each sample and each family;
2) The second step (lines 8–22) calculates the TF indicator for each callback behaviour CB_k that has been selected as one of the attributes in the feature vector FV_i. The variable *appear* records the number of the applications where CB_k may appear. Remember that, to determine whether CB_k belongs to an application $a_{i,j}$ (line 14), we should first compute the similarity between CB_k and every callback behaviour of $a_{i,j}$ (Section IV-B), and then observe whether the similarity is greater than the threshold 90%. If so, it suggests that CB_k belongs to $a_{i,j}$, and then *appear* is increased.
3) The final step (lines 23–25) calculates the IDF indicator for each CB_k, where $inFamily(\cdot, \cdot)$ is a two-dimensional array, which records whether a callback behaviour appears in a family.

D. CONSTRUCT AN 1-NN CLASSIFIER

We construct an 1-NN classifier to distinguish an unknown sample into its predefined family. To this end, we shall first process the unknown sample to a feature vector, then compute the distances between this sample and every malware family, and assign the nearest one as its family.

Algorithm 2 Calculate the Feature Vectors for 65 Families

Input:
A matrix of malware samples: $\mathcal{M}_{65\times 20}$. Each row of $\mathcal{M}$ is a malware family $\mathcal{F}_i = [a_{i,1}, \ldots, a_{i,20}]$ which includes 20 samples.
Output:
The family feature vectors FV_i for each family $\mathcal{F}_i (1 \le i \le 65)$. Each FV_i is a 3211-dimensional vector.

```
FBeh(F_i) = ∅ for all 1 ≤ i ≤ 65;
for i = 1, ..., 65 do
    for j = 1, ..., 20 do
        ABeh(a_{i,j}) = fuzzy processing of regular expressions of a_{i,j};
        FBeh(F_i) = FBeh(F_i) ∪ ABeh(a_{i,j});
    end
end
foreach F_i (1 ≤ i ≤ 65) do
    foreach CB_k ∈ FBeh(F_i) do
        appear = 0;
        inFamily(CB_k, F_i) = 0;
        if CB_k is one of the selected attribute then
            for j = 1, ..., 20 do
                if CB_k ∈ ABeh(a_{i,j}) then
                    appear = appear + 1;
                    inFamily(CB_k, F_i) = 1;
                end
            end
        end
        TF(CB_k, F_i) = appear / |FBeh(F_i)|;
    end
end
foreach CB_k in the selected attributes do
    IDF(CB_k, C) = log 65 / (1 + Σ_{j=1}^{65} inFamily(CB_k, F_j));
end
```

Processing an unknown sample, say a, is much similar to that of malware families, and shall involve the following steps:

1) Extract the regular expressions of callbacks of a, and apply the fuzzy processing rules to these expressions to obtain ABeh(a);
2) For each callback behaviour $CB_k \in$ ABeh(a), if CB_k is one of the selected attributes, then we shall compute its weight using TF-IDF indicators:

$$w_k = \mathrm{TF}(CB_k, a) \times \mathrm{IDF}(CB_k, \mathcal{C}).$$

3) Contruct the feature vector $V(a) = [w_1, w_2, \ldots, w_{3211}]$ for application a.

To measure the *similarity* or *distance* between an unknown sample (i.e., the vector $V(a)$) and a family (i.e., the vectors FV_i), we adopt the traditional *cosine similarity* [23] between two vectors.

The cosine similarity between the vectors

$$\boldsymbol{d} = (d_1, d_2, \ldots, d_k)$$

and

$$\boldsymbol{e} = (e_1, e_2, \ldots, e_k)$$

is defined as

$$sim(\boldsymbol{d}, \boldsymbol{e}) = \frac{\boldsymbol{d}\cdot\boldsymbol{e}}{\|\boldsymbol{d}\|\,\|\boldsymbol{e}\|} = \frac{\sum_{i=1}^{k} d_i\, e_i}{\sqrt{\sum_{i=1}^{k} d_i^2}\sqrt{\sum_{i=1}^{k} e_i^2}}.$$

Then we compare these cosine similarities $sim(V(a), FV_i)$ $(1 \le i \le 65)$ one by one and determine a to a family $\mathcal{F}_I$ with the largest similarity, that is,

$$sim(V(a), FV_I) = \max_{1\le i\le 65} sim(V(a), FV_i).$$

VI. PROTOTYPE AND EXPERIMENTS

A. IMPLEMENTATION

The prototype of our approach is based on Soot [18] and Sklearn library [24]. Soot is a Java bytecode optimization framework formerly developed by Sable Research Group of McGill University. It provides a variety of analysis frameworks for Java programs, such as IFDS/IDE dataflow analysis, Call graph construction and Point-to analysis. In this article, we only use Soot to generate (interprocedural) control flow graphs for every implemented class methods or callbacks. Sklearn is a python-based third-party machine learning library, which helps us train the 1-NN classifier. The overview and implementation of our approach have been shown in aforementioned Fig. 1.

B. TRAINING DATASETS

Our work is largely based on a substantial dataset of real-world Android OS malware samples. We collected the dataset from Drebin project [8], Virus Share [25], FalDroid [12] and Android Malware Genome Project [1], which cover the majority of malware families and include a variety of infection techniques and payload functionalities. Table 4 lists the datasets adopted in our paper.

More than 90% of malware samples are smaller than 5 MB, and approximately 3% of the samples are larger than 10 MB. For the purpose of this article, we discarded a number of encrypted samples and the malware families that contain only one sample, resulting in a final dataset of 3270 malware samples grouped into 65 families, from which a total of 23568 callback behaviours have been extracted. The dataset, implementation, and immediate experimental results have been uploaded to Github and Baidu Cloud Disk (see Subsection I-B).

C. EXPERIMENTAL EVALUATION

Our evaluation intends to investigate the following research questions:

RQ1 Why do we select *CB*s (Callback Behaviours) as a predominant feature to do the multifamily classification task?

TABLE 4. Sample dataset.

Dataset	#Samples	#Families	Average Size (MB)
Genome [1]	1,247	22	1.3
Drebin [8]	5,513	132	1.3
FalDroid-II [12]	643	43	2.0
VirusShare [25]	1,160	80	1.8

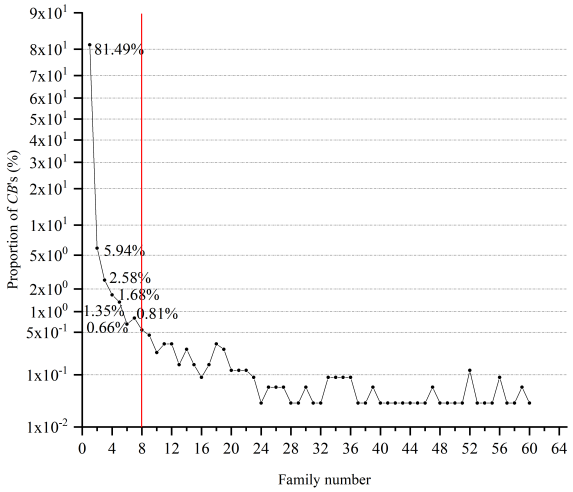


FIGURE 6. Distribution of *CB*s over 65 families.

In other words, why do *CB*s offer a higher discrimination power for the classification?

RQ2 Approximately how many *CB*s are in each family? How many common callback behaviours (i.e., $|\text{cFBeh}(\mathcal{F}_i)|$) and discriminative callback behaviours (i.e., $|\text{dFBeh}(\mathcal{F}_i)|$) are in each family?

RQ3 How well does the 1-NN classifier perform when being applied to the test dataset?

RQ4 How does our approach compare against other state-of-the-art approaches or tools in terms of precision?

In the sequel we address each research question in detail.

RQ1: High discrimination power of *CB*s

We first investigate the distribution of all 23568 *CB*s over 65 malware families, as shown in Fig. 6. The result shows that 81.49% of the *CB*s only appear in individual malware families, indicating that if a *CB* appears in a family, it is unlikely to appear in the other families simultaneously. The proportion of the *CB*s that appear simultaneously in two families drops to 5.94%, and less than 5% of *CB*s distribute in more than 9 families at the same time. From this distribution, it follows that callback behaviours can serve as a strong candidate to distinguish most of the malware families from each other.

RQ2: Number of *CB*s in each family

We next investigate how many *CB*s are in each malware family. This exploration will provide an intuitive insight into why we choose callback behaviours as a feature to train the classifier. The distribution of the number of *CB*s over the families is illustrated in Fig. 7.

It shows that the number of *CB*s (i.e., $|\text{FBeh}(\mathcal{F}_i)|$) has a wide range in different families, thus it seems no sense to do with the average number of *CB*s within a family. This fact might be caused by the wide spread and popularity of a certain family. For example, the families such as DroidKungFu and Plankton, commonly appear in many repackaged malware, so the total number of *CB*s in these families exhibits a sharp increase.

TABLE 5. Error rate of our 1-NN classifier.

Error rate=incorrectly classified samples / total samples(%)			
AccuTrack	0/4 (0.00%)	Jifake	0/10 (00.00%)
Adrd	3/16 (18.75%)	Kidlogger	0/5 (00.00%)
AnserverBot	0/2 (00.00%)	Kmin	1/17 (5.88%)
Aks	0/4 (00.00%)	Lemon	0/4 (00.00%)
BaseBridge	2/119 (1.68%)	Mania	0/5 (00.00%)
BeanBot	0/3 (00.00%)	Mobilespy	0/5 (00.00%)
Biige	0/3 (00.00%)	MobileTx	0/11 (00.00%)
Boxer	0/14 (00.00%)	Mobinauten	0/5 (00.00%)
Coogos	0/4 (00.00%)	Nandrobox	0/8 (00.00%)
Copycat	0/5 (00.00%)	Nickspy	0/4 (00.00%)
Cosha	0/6 (00.00%)	Nyleaker	0/10 (00.00%)
Dougalek	0/5 (00.00%)	Opfake	9/263 (3.42%)
DroidDream	1/19 (00.00%)	Penetho	0/7 (00.00%)
DroidKungFu	3/291 (1.03%)	Placms	0/5 (00.00%)
DroidSheep	0/5 (00.00%)	Plankton	7/271 (2.58%)
EICAR-Test-File	0/3 (00.00%)	RootSmart	0/4 (00.00%)
ExploitLinuxLotoor	1/7 (14.28%)	SeaWeth	0/5 (0.0)%
FaceNiff	0/4 (00.00%)	SendPay	1/3 (33.33%)
FakeDoc	2/45 (4.44%)	SerBG	0/6 (00.00%)
FakeInstaller	5/378 (1.32%)	SMSreg	0/16 (00.00%)
Fakelogo	0/7 (00.00%)	SMSZombie	0/3 (00.00%)
Fakengry	0/5 (00.00%)	Spitmo 0/3	(00.00%)
FakePlayer	0/9 (00.00%)	SpyHasb	0/2 (00.00%)
FakeRun	0/9 (00.00%)	Stealer	0/9 (00.00%)
FakeTimer	0/8 (00.00%)	Stiniter	0/3 (00.00%)
FoCobers	0/8 (00.00%)	Trackplus	0/5 (00.00%)
Gappusin	0/6 (00.00%)	TrojanSMS.Hippo	0/9 (00.00%)
Geinimi	2/22 (9.09%)	Typstu	0/4 (00.00%)
GinMaster	7/137 (5.11%)	Vdloader	0/5 (00.00%)
Glodream	0/8 (00.00%)	Xsider	0/7 (00.00%)
Hamob	0/15 (00.00%)	Yzhc	0/15 (00.00%)
Iconosys	0/47 (00.00%)	Zitmo	0/5 (00.00%)
Imlog	0/18 (00.00%)	**Global**	44/1970 (2.23%)

The distributions of the common *CB*s (i.e, $|\text{cFBeh}(\mathcal{F}_i)|$) and the predominant discriminative *CB*s (i.e, $|\text{dFBeh}(\mathcal{F}_i)|$) of families have been shown previously in Fig. 4 (see Section IV-D). The result also shows that the similarity-based fuzzy match extracts more callback behaviours than the plain-text match. For example, 24 of 65 families have empty common callback behaviours with the plaintext match approach; on the contrary, only 1 of 65 families has no common callback behaviours with the similarity-based approach.

RQ3: Performance of the 1-NN classifier

In all samples of the dataset, 25% of them are selected as the test set and are applied to the 1-NN classifier for family classification. The experimental results for each family are listed in Table 5. It shows that the average error rate of the 1-NN classifier is only 2.23%, achieving a good result in terms of classification precision.

RQ4: Comparison with state-of-the-art approaches

To confirm the effectiveness of our approach, we mainly compare our approach with two recent typical works: DENDROID [17] and FalDroid [12], since the both show the state-of-the-art performance and have given comprehensive experimental results that can be readily used for conducting comparison experiments. DENDROID collects 33 malware families, and the average precision is about 94%; FalDroid collects 36 families, and the average precision is 94.2%.

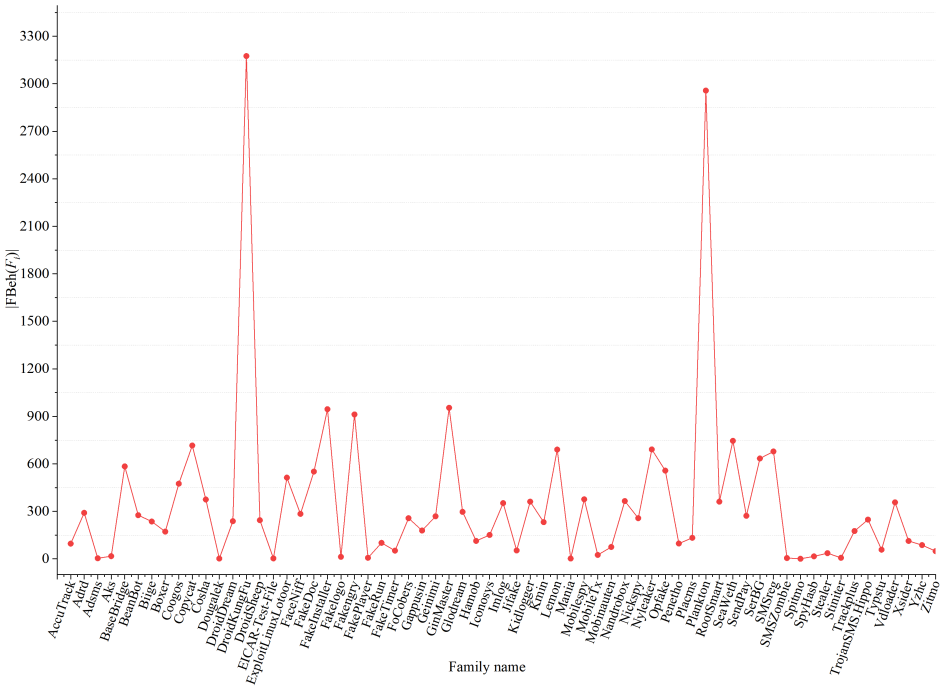


FIGURE 7. The number of *CBs* (i.e., $|\text{FBeh}(\mathcal{F}_i)|$) in each family.

Our work falls all malware samples into 65 families, and the average precision is 97.8%, achieving a better average classification result than the both.

Furthermore, we compare the detection precisions for each malware family. Considering DENDROID, FalDroid and our work have different malware families, we select a collection of common malware families in these three works, and apply our dataset to them to compare the classification results with respect to each individual family. The comparison results are shown in Figure 8, showing that the accuracy of our approach outperforms against DENDROID and FalDroid in general. The reason for such a superiority is that our approach takes the *contexts* information into account when selecting features—the *callbacks* that are predefined in various components are considered as the basic behaviour units. Callbacks usually serve as the entry points of program execution, therefore selecting them as features will facilitate identifying the behaviour patterns of families. Additionally, callbacks retain a more coarse-grained granularity than class methods, this further produces a benefit of dimensionality reduction for the classification.

VII. RELATED WORK

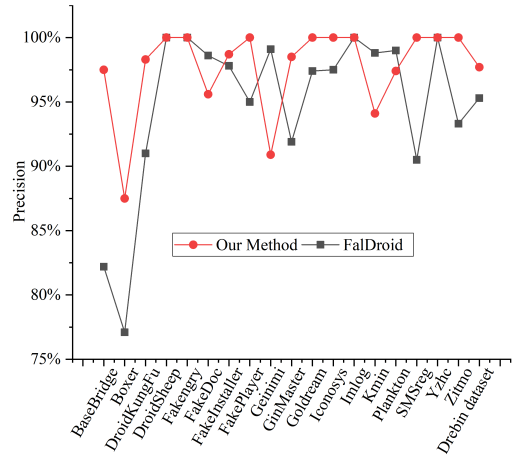
The methods and tools for automatic Android malware detection have been intensively studied over the recent years. There are two basic kinds of detection techniques—*static* and *dynamic* approaches [4], [27]–[30]—according to whether execution of a specimen is required in the process of detection. The static approaches can be further divided into *whitebox* and *blackbox* ones. Note that, according to this taxonomy, our work falls into the category of *static* and *blackbox* approaches. In what follows, we focus on the static and blackbox approaches, and mainly overview their basic ideas and virtues and limitations.

A. WHITEBOX VS. BLACKBOX STATIC APPROACHES

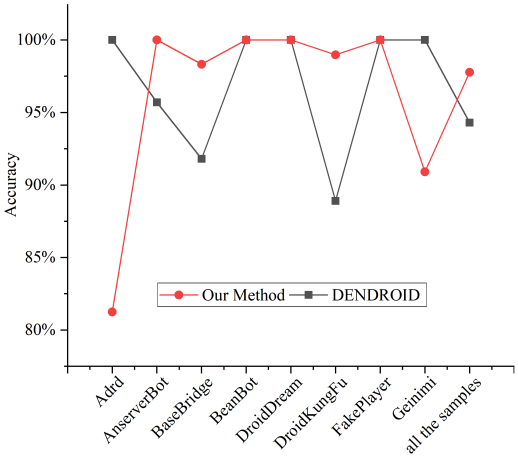
Static analysis approaches are able to directly analyse the text of an applications without executing it at all. According to whether taking the states of a program into consideration, the static approaches are divided into two categories: whitebox [31], [32], [34] and blackbox.

The whitebox approaches require to investigate the structures and semantics of programs, and detect misbehaviours by using traditional program analysis techniques [33] (such as dataflow analysis) or by discovering certain semantic inconsistencies in programs. The most typical whitebox work is FlowDroid [31], a static taint analysis approach and tool-support, which concentrates on discovering potential information leakages from private sources to public sinks. The major benefit of the whitebox approaches is that they are able to precisely reveal the working process of malbehaviours; but on the other hand, they are vulnerable to the high complexity and poor scalability, usually caused by the nature of static program analysis techniques.

On the contrary, the blackbox methods do not tangle with the details of internal structure of a program, but focus on the external features of the program, and employ some kinds of statistical methods to detect misbehaviours. With a large number of malware samples being available and the rapid development of statistical machine learning techniques, the blackbox methods have been found promising for the detection of Android malware.



(a) Compare with FalDroid



(b) Compare with DENDROID

FIGURE 8. Comparison results with the state-of-the-art works.

B. MACHINE LEARNING-BASED MULTIFAMILY CLASSIFICATION APPROACHES

From the viewpoint of machine learning techniques, malware detection can be considered as a supervised binary or multi-family classification problem. The main challenges of these solutions consist in how to *select* and *generate* representative features, and how to resist code obfuscation and familial variants. Permissions, APIs, control flow graphs and their associated contexts are frequently selected as features [2], [3], [8], [9], [12], [13], [17], [34].

Our work mostly inspired by DENDROID [17] and FalDroid [12]. DENDROID selects code structures (the sequences of program statements) of class methods as features, and adopt classical text mining and standard Vector Space Model to classify malware samples into families. FalDroid [12] proposed a graph-based approach to Android malware familial classification. It constructs frequent subgraphs to represent the common behaviours of malware samples, and performs graph match with the weighted-sensitive-API-call-based approach to compete against polymorphic variants.

Our work improves the both studies from three aspects. First, we select the regular expressions of callbacks as features. Since callbacks act as the entry points of an Android application, their behaviours tend to offer more semantic information than those of general class methods; second, the proposed graph transformation of iCFGs (Section III) is able to effectively tackle certain forms of code obfuscation, such as renaming user-defined functions, nested calls and control flow obfuscation via filtering out the security-irrelevant nodes and edges from the iCFGs; and finally, we propose the two-step fuzzy processing strategy to compete against polymorphic variants. The highlight of this strategy is of the substitution operation that widens the meaning of sensitive API calls, achieving a better resilience to familial variants than DENDROID and FalDroid.

Our work is also closely related to DroidSIFT [14] and DESCRIBEME [15], both of which utilize WC-ADGs (Weighted Contextual API Dependency Graphs) as program semantics to construct feature sets. A WC-ADG contains various key semantic-level behaviour aspects of an Android malware sample, such as API dependency, context and data dependency. The main complexity of the work derives from the construction of WC-ADGs, which involves diverse static analysis techniques, including the (forward and backward) dataflow analysis, constant analysis and generation of program dependency graphs. Comparison with the both work, our approach only involves CFGs and API calls; the construction of CFGs and extraction of behaviour aspects are straightforward and efficient, leading to an easier implementation than these work.

To better fight against familial variants, a novel method called *Artificial Malware-based Detection* is proposed [35], which generates new malware patterns using the genetic algorithm from the patterns that have been found from existing malware, for the sake of accommodating the emergence of unseen variants.

The study [10] proposes an approach to selecting the fingerprints (i.e., the signatures) of malware families. It uses the Fisher discriminative criteria to rank the features, and devises a frequency-based feature elimination algorithm which is parallel with the TD-IDF algorithm in our work. This study selects suspicious API calls, permissions and hardware components as features for classification; however our approach takes API calls and their calling sequences as features that hold richer semantic information than individual API calls.

EC2 [36] is a hybrid approach that considers both static and dynamic features. The static features include authors, structures and permissions, and dynamic ones include n-grams of read and write operations, system operations, network operations and SMS operations. It combines the results of supervised classification methods and unsupervised clustering methods together to classify families. The major limitation of this work is that it fails to fully address the issue of code obfuscation and familial variants, thus degrading the accuracy of classification.

VIII. CONCLUSION

This article proposes a static multifamily detection approach for Android malware based on text mining and machine learning techniques. The most significant aspects of our approach are of twofold: 1) the graph reduction transformation of iCFGs (Section III), and 2) the two-step fuzzy processing strategy for the regular expressions (Section IV). The reduction transformation of iCFGs merely reserves the security-sensitive API nodes and their calling sequences. In this way we can tackle various forms of obfuscation such as renaming user-defined functions, nested calls and control flow structure obfuscation. The two-step fuzzy processing strategy first performs a substitution operation on the regular expressions to raise up the level of abstraction and cover a broader range of behaviour patterns, then relaxes a plaintext match into a fuzzy match to resist potential variations to a family behaviour pattern.

At present, we only discovered the common behaviour patterns (i.e., the regular expressions of API calls) of malware families, but it is unclear yet how these patterns are stimulated and behaved to accomplish a malicious attack. This desire will motivate us to further investigate the intentions of behaviour patterns and explore the intricate working mechanisms of malware families in the near future. Another, ICCs (Inter-Component Communications), especially implicit ICCs, have not been involved in this article; this limitation will drive us to improve the construction of iCFGs by fully considering implicit calling relations among components.

REFERENCES

[1] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, May 2012, pp. 95–109, doi: 10.1109/SP.2012.16.

[2] J. Zhang, Z. Qin, H. Yin, L. Ou, and K. Zhang, "A feature-hybrid malware variants detection using CNN based opcode embedding and BPNN based API embedding," *Comput. Secur.*, vol. 84, pp. 376–392, Jul. 2019, doi: 10.1016/j.cose.2019.04.005.

[3] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, "Yes, machine learning can be more secure! A case study on Android malware detection," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 4, pp. 711–724, Jul. 2019, doi: 10.1109/TDSC.2017.2700270.

[4] L. Chen, M. Zhang, C.-Y. Yang, and R. Sahita, "POSTER: Semi-supervised classification for dynamic Android malware detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Oct. 2017, pp. 2479–2481, doi: 10.1145/3133956.3138838.

[5] L. Chen, S. Hou, and Y. Ye, "SecureDroid: Enhancing security of machine learning-based detection against adversarial Android malware attacks," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, New York, NY, USA, Dec. 2017, pp. 362–372, doi: 10.1145/3134600.3134636.

[6] J. Wang, Q. Jing, J. Gao, and X. Qiu, "SEdroid: A robust Android malware detector using selective ensemble learning," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Seoul, South Korea, May 2020, pp. 1–5.

[7] N. Xie, F. Zeng, X. Qin, Y. Zhang, M. Zhou, and C. Lv, "RepassDroid: Automatic detection of Android malware based on essential permissions and semantic features of sensitive APIs," in *Proc. Int. Symp. Theor. Aspects Softw. Eng. (TASE)*, Guangzhou, China, Aug. 2018, pp. 52–59, doi: 10.1109/TASE.2018.00015.

[8] D. Arp, M. Spreitzenbarth, M. Häbner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Los Angeles, CA, USA, 2014, pp. 1–15, doi: 10.14722/ndss.2014.23247.

[9] X. Liu, Q. Lei, and K. Liu, "A graph-based feature generation approach in Android malware detection with machine learning techniques," *Math. Problems Eng.*, vol. 2020, May 2020, Art. no. 3842094, doi: 10.1155/2020/3842094.

[10] N. Xie, X. Wang, W. Wang, and J. Liu, "Fingerprinting Android malware families," *Frontiers Comput. Sci.*, vol. 13, no. 3, pp. 637–646, Jun. 2019, doi: 10.1007/s11704-017-6493-y.

[11] BJ. Kang, SY. Yerima, S. Sezer, and K. McLaughlin, "N-gram opcode analysis for Android malware detection," *Int. J. Cyber Situational Awareness*, vol. 1, no. 1, pp. 231–255, 2016. [Online]. Available: https://arxiv.org/abs/1612.01445

[12] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018, doi: 10.1109/TIFS.2018.2806891.

[13] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, Florence, Italy, May 2015, pp. 303–313, doi: 10.1109/ICSE.2015.50.

[14] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2014, pp. 1105–1116, doi: 10.1145/2660267.2660359.

[15] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for Android apps," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2015, pp. 518–529, doi: 10.1145/2810103.2813669.

[16] I. Kononenko, "Estimating attributes: Analysis and extensions of RELIEF," in *Proc. Eur. Conf. Mach. Learn.* Berlin, Germany: Springer, 1994, pp. 171–182, doi: 10.1007/3-540-57868-4_57.

[17] 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," *Exp. Syst. Appl.*, vol. 41, no. 4, pp. 1104–1117, 2014, doi: 10.1016/j.eswa.2013.07.106.

[18] *Soot*. Accessed: Sep. 1, 2018. [Online]. Available: https://github.com/Sable/soot

[19] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory: Languages, and Computation*. Reading, MA, USA: Addison-Wesley, 1990.

[20] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Los Angeles, CA, USA, 2014, pp. 1125–1139, doi: 10.14722/ndss.2014.23039.

[21] M. Mohri, "Edit-distance of weighted automata: General definitions and algorithms," *Int. J. Found. Comput. Sci.*, vol. 14, no. 06, pp. 957–982, Dec. 2003, doi: 10.1142/S0129054103002114.

[22] M. Mohri, *Weighted Automata Algorithms*. Berlin, Germany: Springer, 2009, pp. 213–254.

[23] C. Aggarwal, *Machine Learning for Text*. New York, NY, USA: Springer, 2018.

[24] *SKlearn*. Accessed: Feb. 18, 2019. [Online]. Available: https://sklearn.apachecn.org/

[25] *Virus Share*. Accessed: Sep. 1, 2017. [Online]. Available: https://virusshare.com

[26] *360 Security Guard*. Accessed: Sep. 1, 2017. [Online]. Available: https://www.360.cn/

[27] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, Jun. 2014, doi: 10.1145/2619091.

[28] 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, pp. 10107–10120, 2018, doi: 10.1109/ACCESS.2018.2804321.

[29] M. Backes, S. Bugiel, O. Schranz, P. Von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android runtime instrumentation and security toolkit," in *Proc. IEEE Eur. Symp. Secur. Privacy*, Paris, France, Apr. 2017, pp. 481–495, doi: 10.1109/EuroSP.2017.43.

[30] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for Android RunTime," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 331–342, doi: 10.1145/2976749.2978343.

[31] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, Edinburgh, U.K., 2013, pp. 259–269, doi: 10.1145/2594291.2594299.

[32] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 576–587, doi: 10.1145/2635868.2635869.

[33] F. Hielson, H. R. Nielson, and C. H. ankin, *Principles of Program Analysis*. Berlin, Germany: Springer, 2005, doi: 10.1007/978-3-662-03811-6.

[34] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Phoenix, AZ, USA, 2014, pp. 1329–1341, doi: 10.1145/2660267.2660357.

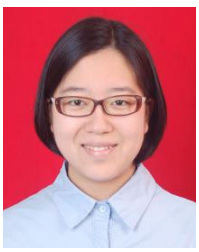
[35] M. Jerbi, Z. C. Dagdia, S. Bechikh, and L. B. Said, "On the use of artificial malicious patterns for Android malware detection," *Comput. Secur.*, vol. 92, May 2020, Art. no. 101743, doi: 10.1016/j.cose.2020.101743.

[36] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian, "EC2: Ensemble clustering and classification for predicting Android malware families," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 2, pp. 262–277, Mar. 2020, doi: 10.1109/TDSC.2017.2739145.

XIAOJIAN LIU received the B.S. and M.S. degrees in mathematics from Nanjing University, Nanjing, China, in 1990 and 1997, respectively, and the Ph.D. degree from Xidian University, Xi'an, China, in 2004. From 2004 to 2006, he was a Postdoctoral Fellow with the Department of Computer Science, Northwestern Polytechnic University. In 2005, he was a Visiting Fellow with the United Nations University International Institute for Software Technology (UNU-IIST), Macau. He is currently an Associate Professor with the College of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an. His research interests include software security and program analysis.

XI DU was born in Yongle, Zhen'an, Shangluo, Shaanxi, China. She received the bachelor's degree in IoT from the Xi'an University of Science and Technology, in 2018, where she is currently pursuing the master's degree in the major of computer system architecture. Her research interest includes static detection approaches for Android malware.

QIAN LEI was born in Zhongzhang, Liyang, Xianyang, Shaanxi, China, in 1995. She received the bachelor's degree from the Xi'an University of Science and Technology, in 2017, where she is currently pursuing the master's degree. During her graduate studies, she has been engaged in the research of static detection approaches for Android malware and has published several journal and conference papers on this topic.



KEHONG LIU was born in Xi'an, Shaanxi, China, in 1999. She is currently pursuing the bachelor's degree with the Xi'an University of Science and Technology. During her undergraduate studies, she has been engaged in the research project of Android malware detection and has published several journal and conference papers on this topic.