CTDroid: Leveraging a Corpus of Technical Blogs for Android Malware Analysis

Ming Fan, Xiapu Luo, Jun Liu, Chunyin Nong, Qinghua Zheng, and Ting Liu

Abstract—**The rapid growth of Android malware results in a large body of approaches devoted to malware analysis by leveraging machine learning algorithms. However, the effectiveness of these approaches primarily depends on the manual feature engineering process, which is time-consuming and labor-intensive based on expert knowledge and intuition. In this paper, we propose an automatic approach that engineers informative features from a corpus of Android malware related technical blogs, which are written in a way that mirrors the human feature engineering process. However, there are two main challenges. First, it is difficult to recognize useful knowledge in the magnanimity information of thousands of blogs. To this end, we leverage natural language processing techniques to process the blogs and extract a set of sensitive behaviors that might do harmful activities to users potentially. Second, there exists a semantic gap between the extracted sensitive behaviors and the programming language. To this end, we propose two semantic matching rules to match the behaviors with concrete code snippets such that the apps can be tested experimentally. We design and implement a system called CTDroid for malware analysis, including malware detection (MD) and familial classification (FC). After the evaluation of CTDroid on a large scale of real malware and benign apps, the experimental results demonstrate that CTDroid can achieve 95.8% true positive rate with only 1% false positive rate for MD and 97.9% accuracy for FC. Furthermore, our proposed features are more informative than those of state-of-the-art approaches.**

Index Terms—**Android malware, informative feature, natural language process (NLP), technical blog.**

I. Introduction

NOWADAYS, Android malware has posed great threats to smartphone users, such as stealing personal information, connecting to remote command and control servers, and sending premium messages. A recent study conducted by Qihoo reported that about 7.6 million malware were detected in 2017 [1].

A large body of research has thus studied approaches for analyzing Android malware. These approaches increasingly depend on machine learning techniques, which engineer multiple features and train classifiers to detect whether a given application (app) is malicious, and if so, which malware family it belongs to. The existing proposed features can be roughly classified in twofolds: First, string-based features, which are mainly composed of request permissions[1] [2], intents[2] [3], application programming interface (API) calls [4], and other components of Android operation system [5]. Second, structure-based features, which are extracted from different kinds of graph models, including control dependency graph [6]–[9] and function call graph [10]–[13], through heavily inspecting of app code.

The effectiveness of the above approaches primarily depends on the manual feature engineering process, which is time-consuming and labor-intensive based on human knowledge and intuition. Specifically, to perform malware analysis with high performance, the researchers need to manually inspect the malicious activities of malware samples and summarize the hypotheses about common behaviors that malware share but benign apps do not. Furthermore, the summarized hypotheses might vary from different inspected malware samples, thus constructing different feature spaces for different datasets.

Therefore, in this paper, we aim to automatically engineer informative features from existing knowledge learned by experts. Specifically, we mine *sensitive behaviors*, behaviors that might do harmful activities to users potentially, from a corpus of Android malware related technical blogs. The technical blogs in

[1]Android permission control is one of the major Android security mechanisms. Android permissions are requested by apps before the apps can use certain system data and features.

[2]An intent is a bundle of information describing a desired action, including the data to be acted upon, the category of a component that should perform the action, and other pertinent instructions.

Manuscript received June 1, 2018; revised October 18, 2018 and April 4, 2019; accepted June 19, 2019. This work was supported by the Fundamental Research Funds for the Central Universities, in part by the National Key R&D Program of China under Grant 2016YFB1000903, in part by the National Natural Science Foundation of China under Grant 61532004, Grant 61532015, Grant 61632015, Grant 61602369, Grant U1766215, Grant 61772408, Grant 61702414, Grant 61833015, in part by the Innovative Research Group of the National Natural Science Foundation of China under Grant 61721002, in part by the Ministry of Education Innovation Research Team (IRT_17R86), consulting research project of Chinese academy of engineering "The Online and Offline Mixed Educational Service System for 'The Belt and Road' Training in MOOC China" and Project of China Knowledge Centre for Engineering Science and Technology, and in part by the Hong Kong RGC Project (No. PolyU 152279/16E, 152223/17E, CityU C1008-16G). Associate Editors: Zijiang Yang; Bei Bu. *(Corresponding author: Ting Liu.)*

M. Fan is with the School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an 710049, China, with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, and also with the MOEKLINNS Lab, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: fanming.911025@stu.xjtu.edu.cn).

X. Luo is with the Department of Computing, The Hong Kong Polytechnic University, 999077, China (e-mail: luoxiapu@gmail.com).

J. Liu, C. Nong, and Q. Zheng are with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China, and also with the MOEKLINNS Lab, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: liukeen@xjtu.edu.cn; cynong@foxmail.com; qhzheng@xjtu.edu.cn).

T. Liu is with the School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an 710049, China, and also with the MOEKLINNS Lab, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: tingliu@xjtu.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2019.2926129

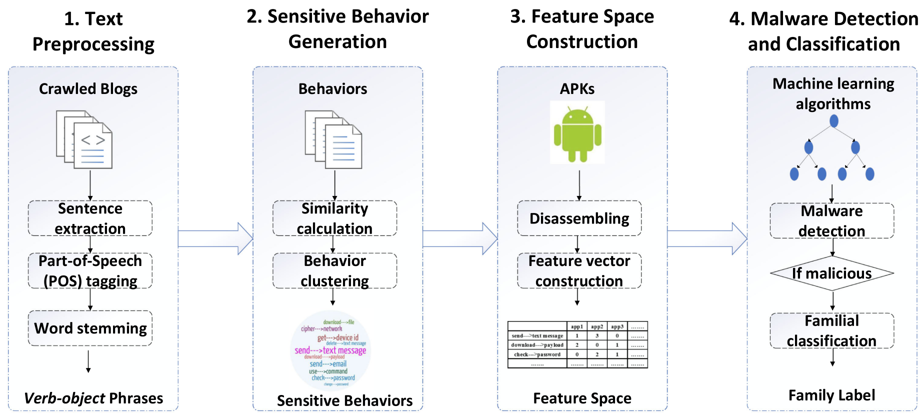


Fig. 1. Overview architecture of CTDroid that contains four main procedures: (1) Text preprocessing (Section II-A). (2) Sensitive behavior generation (Section II-B). (3) Feature space construction (Section II-C). (4) MD and classification (Section II-D).

this paper refer to the entries or articles in the security websites. We choose the technical blogs as our knowledge source because they are written in a way that mirrors the human feature engineering process and they are usually public online in time. Then, we use the extracted sensitive behaviors to guide the automatic informative feature engineering processing. However, there are two main challenges in this paper.

First, it is a key challenge to automatically recognize the harmful activities and mine sensitive behaviors in the magnanimity information of thousands of technical blogs. For example, for the sentence: *"These instructions can be used to open a web page, call a phone number, or send an SMS text message to a premium number. [14],"* it is easy for the researchers to obtain the knowledge that there are three sensitive behaviors marked with an underline for the instructions. However, this conclusion is based on the prior knowledge of research in the world, since the sentence does not provide sufficient linguistic clues that such three behaviors might do harmful activities. Therefore, there is a semantic gap between the natural language in technical blogs and sensitive behaviors.

Second, there also exists a semantic gap between the sensitive behaviors and the programming language. Therefore, it is hard to directly utilize the sensitive behaviors for malware analysis with machine learning algorithms. For example, even when we know that *send an SMS text message* is a sensitive behavior, we are still unable to directly identify how does a given app perform such sensitive behavior in their thousands of lines of code.

To overcome the first challenge, we leverage natural language processing (NLP) techniques to parse the contents in blogs into a uniform structure, verb–object phrase (e.g., "send—> text message"). Then, we propose a clustering-based approach to extract frequent behaviors that have close relations with Android system and regard them as sensitive behaviors. For the second challenge, we propose two semantic matching rules to bridge the gap between the sensitive behavior and the programming language based on the analysis of descriptions of Android concrete features (i.e., permissions, API calls, and intents), as well as the keywords in the app code.

We implement these ideas in a system called CTDroid to construct a set of informative features. With known machine learning algorithms, we train classifiers and perform malware detection (MD) and familial classification (FC). After applying CTDroid on a large scale of real malware and benign apps, we find that our constructed features exhibit impressive malware analysis performance. In summary, our major contributions include the following.

1) We propose techniques that summarize the existing knowledge contained in magnanimity information of natural language documents and generate a novel type of features presented as verb–objective phrases that are easy to understand.
2) We propose two semantic matching rules that bridge the gap between the phrase-based features and programming language.
3) We design and implement CTDroid, an automatic feature engineering system. By using CTDroid, we construct a set of informative features that can be utilized for Android MD and FC.
4) We conduct extensive experiments to evaluate CTDroid on a large scale of real malware and benign apps. The experimental results show that CTDroid can achieve 95.8% true positive rate (TPR) with only 1% false positive rate (FPR) for MD and 97.9% accuracy for FC. Furthermore, our proposed features are more informative than those of state-of-the-art approaches.

The rest of this paper is organized as follows. Section II details the methodology of CTDroid. Section III reports the experimental results. After discussing the limitations of CTDroid in Section IV, we introduce the related work in Section V, and Section VI concludes this paper.

II. Methodology

Fig. 1 illustrates the overview architecture of CTDroid, which is consisted of four main procedures.

In the *text preprocessing* procedure, at first, a corpus of technical blogs are crawled from websites and used as the input of our system. Then, by applying NLP techniques, including sentence extraction, part-of-speech (POS) tagging, and word stemming, the contents in blogs are parsed into a set of behaviors that are represented as verb–object phrases.

In the *Sensitive behavior generation* procedure, since not every behavior extracted from the blogs is significant for malware analysis, the extracted behaviors are grouped into a set of clusters and the frequent behaviors that have close relations with Android system are identified as the sensitive behaviors.

In the *Feature space construction* procedure, each defined sensitive behavior is regarded as a feature. Then, two matching rules are proposed to construct a feature space, where each app is represented as a feature vector.

In the *MD and classification* procedure, with known machine learning algorithms, different classifiers are generated for the purposes of MD and FC.

A. Text Preprocessing

There are thousands of technical blogs on the web. It is neither effective (need an expert understanding of Android system) nor efficient (too much knowledge to learn) to carefully read each blog. To better automatically obtain significant knowledge from the blogs, we first transform the semantic meanings of blog contents into a set of behaviors. A behavior is represented as a tuple that consists of a verb and an object, and both of them are indispensable. The steps of extracting behaviors from blogs with NLP techniques are listed as below.

1) Sentence Extraction: Given a technical blog crawled from the website with HTML format, we initially use *jsoup* [15], a Java HTML parser, to extract the contents from the HTML file and remove all non-ASCII symbols. Then, we split the extracted content into a set of sentences with sentence segmentation.

2) POS Tagging: For each extracted sentence, its typed dependency representations of the plain text in the form of *rule(gov, dep)* are extracted by using *Stanford typed dependency parser* [16], [17], a program that works out the grammatical structure of sentences. The *gov* and *dep* denote the governor word and the dependent word, respectively. The *rule* denotes the relation between the *gov* and *dep*. There are many different kinds of *rules* defined in the parser, such as *conj*, *det*, *dobj*, and *nsubjpass*. By carefully reading ten technical blogs, we find that most of the extracted subjects are the malware samples. Therefore, we do not consider the rules of which the *dep* is the subject. Moreover, given that we aim to extract the information that depicts the malicious activities conducted by malware, the rules such as *det* and *conj* that depict the definition and conjunction relationships cannot be used to find the malicious activities. Thus, we only focus on two main types of *rules*: *dobj* and *nsubjpass*. The *dobj* denotes that the *dep* is the (accusative) object of the *gov*. The *nsubjpass* denotes that the *dep* is the syntactic subject of the *gov* in a passive clause.

As listed in Table I, after the decomposition of the plain text, we can get the corresponding typed dependency representations with the *gov* (i.e., "open," "call," and "send") and the *dep* (i.e., "page," "number," and "message"). We construct one behavior for each generated typed dependency representation, where the *gov* is used as the *verb* and the *dep* is used as the *object*. Furthermore, we extend the *verb* and the *object* to their corresponding noun phrases by adding the adjective modifiers and identifying multiword expressions. For example, the *object* "message" is extended to its noun phrase, i.e., "sms text message."

TABLE I
EXAMPLE OF BEHAVIOR EXTRACTION

plain text	"These instructions can be used to open a web page, call a phone number, or send an SMS text message to a premium number." [14]
typed dependency representations	dobj(open, page) dobj(call, number) dobj(send, message)
behaviors	open—> web page call—> phone number send—> sms text message

TABLE II
FOURTEEN SEMANTIC GROUPS AND THEIR REPRESENTATIVE VERBS

Representative verb	**Similar verbs**
send	push, upload
get	return, obtain, collect, gather, gain, access
check	verify, confirm
connect	direct, redirect, open, call, contact
use	invoke, perform, run, execute, activate, conduct
cipher	encrypt, encode
decipher	decrypt, decode
delete	remove, wipe
prevent	abort, restrict, cease, disrupt, intercept
change	replace, modify, alter
set	reset
generate	create, build, make
store	write, remember, impress
download	load, install, deploy

3) Word Stemming: The noun phrases with similar semantic meaning would appear in different variants, such as "a phone number" and "phone numbers." To address this problem, we first remove the stop words, the common words that would appear to be of little value for NLP analysis. The stop words used in our paper is provided by [18], such as "a," "an," and "the". Then, we apply *WordNet* [19] to reduce the words based on their POS tag to their root forms. For example, the object "numbers" in its plural form would be reduced to "number."

Then, given that different verbs may have similar meanings, such as "get" and "return," we regard these verbs as the same one. To this end, we manually construct 14 semantic groups based on a set of commonly used verbs provided by Anton *et al.* [20]. Then, we add their similar verbs returned by *WordNet*. As listed in Table II, each semantic group consists of a set of similar verbs and one representative verb. If the verb of a behavior belongs to one of the semantic group, then it will be replaced with the corresponding representative verb. For example, the behavior "return—> phone number" will be changed to "get—> phone number."

B. Sensitive Behavior Generation

After the text preprocessing of the collected blogs, 208K behaviors are extracted. However, we observe that most of the extracted behaviors present little significance for malware analysis. For example, the behavior "advise—> user" occurs when the researchers give some advice to the users about how to protect their smartphones. However, this behavior has little value for malware analysis. Thus, we propose a clustering-based approach to filter out the useless behaviors and mine the frequent behaviors that have close relations with Android system. These behaviors are regarded as the sensitive behaviors. To this end, we need to first propose an effective and efficient behavior similarity calculation method since there are too many extracted behaviors.

1) Behavior Similarity Calculation: For convenience, we use BH to denote the set of extracted behaviors. $BH = \{bh_i = (verb_i, object_i) | 1 \leq i \leq K\}$, where K is the total number of behaviors. Each behavior bh_i contains a $verb_i$ and an $object_i$. The similarity between two behaviors bh_i and bh_j depend on the similarities between their corresponding $verbs$ and $objects$, which are represented as $sim(verb_i, verb_j)$ and $sim(object_i, object_j)$, respectively. The similarity between bh_i and bh_j is obtained as (1)

$$sim(bh_i, bh_j) = \alpha * sim(verb_i, verb_j) + (1 - \alpha) * sim(object_i, object_j). \tag{1}$$

The parameter α is used to control the weights of the similarity of $verbs$ and $objects$; $0 \leq \alpha \leq 1$. The reason of introducing α is that if the behaviors whose *verbs* are general words, such as "use," "get," and "return," their similarities would mainly rely on the $sim(object_i, object_j)$ rather than $sim(verb_i, verb_j)$. To this end, we assign different weights to the $verbs$ to denote their importance in the similarity calculation. Specifically, if a $verb$ is generally used in our extracted behaviors, then its weight should be low. Thus, we use the inverse document frequency [21] to measure the inverse frequency of $verb$ that appears across all the behaviors. Therefore, the weight of $verb_i$ is calculated as follows:

$$w(verb_i) = \log_2 \frac{K}{Num(bh_{verb_i})} \tag{2}$$

where $Num(bh_{verb_i})$ denotes the number of behaviors that contain $verb_i$. Then, all the $w(verb)$ s are normalized between 0 and 1. Finally, for the $sim(bh_i, bh_j)$, its α is obtained as follows:

$$\alpha = \frac{w(verb_i) + w(verb_j)}{2}. \tag{3}$$

Next, to calculate the similarity between $verbs$ or $objects$ which are actually phrases (phs), we first transform them into a calculable form. Here, we rely on the tool called *Word2Vec* [22]. *Word2vec* takes a large corpus of text as its input and produces a vector space, with each unique word in the corpus being assigned a corresponding vector in the space. In this paper, we collect a 12.2G corpus from Wikipedia [23] and put them into *Word2vec* with the skip-gram model [24]. Each word wd in the corpus is represented as a vector with l dimensions as (4); $l = 100$ in this paper

$$\overrightarrow{vec(wd)} = \langle v_1, v_2, \ldots, v_l \rangle. \tag{4}$$

As introduced in [22], semantic relations among words can be captured via simple vector operation. For example, $\overrightarrow{vec(\text{"}better'')} - \overrightarrow{vec(\text{"}good'')} \approx \overrightarrow{vec(\text{"}faster'')} - \overrightarrow{vec(\text{"}fast'')}$, in which the minus sign denotes vector substraction operation. Leveraging the characteristic of vector operation in *Word2Vec*, we obtain the vector of a phrase ph by the vector adding operation on all the words in ph as (5). The cosine similarity is widely used to find the similarity between two given vectors. Thus, the similarity between two phrases can be calculated with cosine similarity based on (6), in which $||\overrightarrow{vec}||$ is the Euclidean norm of the vector $\overrightarrow{vec}$

$$\overrightarrow{vec(ph)} = \sum_{wd \in ph} \overrightarrow{vec(wd)} \tag{5}$$

$$cosine(\overrightarrow{vec(ph_1)}, \overrightarrow{vec(ph_2)}) = \frac{(\overrightarrow{vec(ph_1)} \times \overrightarrow{vec(ph_2)})}{||\overrightarrow{vec(ph_1)}|| \cdot ||\overrightarrow{vec(ph_2)}||}. \tag{6}$$

Algorithm 1: Clustering of Behaviors.

Input:
$BH = \{bh\}$ // BH denotes the set of extracted behaviors in blogs.
θ // θ denotes the similarity threshold value of adding behaviors into clusters.
ϵ // ϵ denotes the support threshold value of filtering out clusters.
Output:
C // C denotes the set of output clusters and each cluster contains a set of similar behaviors.

$p = 1, c_1 = \{bh_1\}, C = \{c_1\}$
for each $bh_{i,i \neq 1}$ in BH **do**
  $c' = argmax_{c_j \in C} \overline{sim}(bh_i, c_j)$
  if $\overline{sim}(behav_i, c') \geq \theta$ **then**
    $c' = c' \cup \{behav_i\}$
  else
    $p = p + 1, c_p = \{behav_i\}, C = C \cup \{c_p\}$
  end if
end for
for each c_j in C **do**
  if $sup(c_j) < \epsilon$ **then**
    $C.remove(c_j)$
  end if
end for
return C

2) Behavior Clustering: Based on the similarity calculation of behaviors, we mine the frequent behaviors via the clustering of behaviors. Algorithm 1 lists the step of behavior clustering with the input of all generated behaviors $BH = \{bh\}$ and two threshold values, θ and ϵ. θ denotes the similarity threshold between a behavior and a cluster. In other words, if the average similarity between a behavior bh_i with all the behaviors in cluster c_j, represented as $\overline{sim}(bh_i, c_j)$, is higher than θ, then the behavior bh_i is added into the cluster, indicating that the behavior bh_i has very close semantic meanings with those in cluster c_j. ϵ denotes the support threshold value of grouped clusters. If the support value of cluster c_j, represented as $sup(c_j)$ is less than ϵ, we filter out this cluster.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE III
EXAMPLE OF BEHAVIOR CLUSTER

Representative behavior	send—> text message
Behaviors	send—> text message (236) send—> premium text message (3) send—> multiple text message (2) send—> sms text message (2) send—> one text message (2) ......

TABLE IV
ANDROID SYSTEM RELATED CONCEPTS [25]

phone number	photo	imei	password	camera
phone call	time	contact	radio	email
device id	keylock	pin	bookmark	calendar
serial number	network	account	file	package
subscriber id	location	browser	shortcut	screenshot
text message	battery	alarm	wallpaper	bluetooth
microphone	command	permission	activity	wifi

In Algorithm 1, C is initialized with only one cluster $c_1 = \{bh_1\}$ (line 1). Then, all the other behaviors in BH are successively calculated to check whether there exists a cluster in C that the current behavior can be added in (lines 2–9). After that, we filter out the clusters whose support values are less than ϵ (lines 10–14).

After the clustering of behaviors, we can obtain a set of clusters C and each cluster $c \in C$ contains a set of similar behaviors. In each cluster, the behavior with the highest frequency number is selected as the representative behavior $repBh$. The frequency number of a behavior denotes the times of the behavior occurs in BH. Table III lists an example of the generated cluster, in which all behaviors contain the similar semantic meanings of sending text messages. The number attached to the behavior denotes its corresponding frequency number.

To identify the sensitive behaviors, we filter out the behaviors with little significance for malware analysis within two steps.

1) First, we remain the behaviors whose verbs belong to our constructed 14 semantic groups, since most other verbs are too general to identify their concrete actions in app code such as "protect," "alert," and "infect."
2) Second, we remain the behaviors that have close relations with Android system. To this end, we obtain a set of Android system sensitive concepts based on the work of Felt *et al.* [25], in which they conduct research for the user concerns about 99 smartphone risks. As a result, there are 35 sensitive concepts listed in Table IV.

C. Feature Space Construction

After the generation of sensitive behaviors, it is nontrivial to directly utilize the sensitive behaviors for malware analysis with machine learning algorithms due to the semantic gap between the sensitive behaviors and the programming language. To address this challenge, we propose two semantic matching rules by leveraging the descriptions of Android concrete features (i.e., permissions, API calls, and intents), as well as the keywords in the app code.

TABLE V
EXAMPLE OF PERMISSION MATCHING

Sensitive behavior	send—>text message
Permission	SEND_SMS
Description	It allows an application to send SMS messages.
Extracted behavior	send—>sms message (sim: 0.98)

TABLE VI
EXAMPLE OF INTENT MATCHING

Sensitive behavior	check—>package
Intent	PACKAGE_VERIFIED
Description	Send to the system package verifier when a package is verified.
Extracted behavior	verify—>package (sim: 1.0)

1) APK Disassembling: In general, Android apps are written in Java code and they are compiled to Dalvik code (DEX) stored in a file called `classes.dex`. The required resource files and the compiled code are packaged into an APK file. With existing mature disassembling tools, such as *apktool* [26], we are able to obtain the `AndroidManifest.xml` file and the Dalvik code files. The `AndroidManifest.xml` file contains essential information about an app to the Android system, including the requested permissions and intents. It is worth noting that the widely used third-party and advertisement libraries might affect the performance of malware analysis. We filter out these libraries from the Dalvik code by using the blacklist provided by [27], [28].

2) Feature Vector Construction: Basically, the Dalvik code is the main part of an app that we need to match with our sensitive behaviors. Furthermore, existing approaches [2], [29], [30] reveal that permissions and intents are significant for malware analysis. Thus, we also match such concrete features (i.e., permissions and intents) with our sensitive behaviors. Our two matching rules are introduced as follows.

Rule I: Matching With Permissions and Intents: In this paper, 140 permissions and 261 intents are collected from the Android document [31]. However, it is not effective to directly match the permissions and intents with the sensitive behaviors because of the insufficient literal meanings. Therefore, the corresponding descriptions of the permissions and intents are also collected to provide useful information. To match the concrete permissions and intents with given sensitive behavior, the collected descriptions are parsed into behaviors as introduced in Section II-A. In addition, if the name of a permission or an intent consists of a verb and an object, one more behavior is constructed. Then, the extracted behaviors are matched with the given sensitive behavior by using our similarity calculation method. If there exists a similarity that is higher than the preset θ, then we define that the app contains the current sensitive behavior feature.

Tables V and VI list examples of permission matching and intent matching, respectively. The number behinds the extracted behavior denotes its similarity with the sensitive behavior. It is worth noting that in Table VI, since the verb "verify" and the verb "check" belong to the same semantic group listed in Table II,

TABLE VII
EXAMPLE OF API CALL MATCHING

Sensitive behavior	get—>phone number
API	getLine1Number()
Description	It returns the phone number for line 1, for example, the MSISDN for a GSM phone.
Extracted behavior	return—>phone number (sim: 1.0)

```
const-string v3, "content://sms/conversations/"
......
invoke-static {v2}, Landroid/net/Uri;->parse(Ljava/lang/String;)Landroid/net/Uri;
move-result-object v2
const/4 v3, 0x0
const/4 v4, 0x0
invoke-virtual {v0, v2, v3, v4}, Landroid/content/ContentResolver;-
>delete(Landroid/net/Uri;Ljava/lang/String;[Ljava/lang/String;)I
......
```

Fig. 2. Snippets of deleting a text message.

"verify" is replaced with "check" and the similarity between the two behaviors is 1.0.

Rule II: Matching With Dalvik Code: Dalvik code is a human-readable representation of the binary bytecode. From the generated Dalvik code files, we initially extract the method bodies by recognizing the identifies `.method` and `.end method`. In each method body, API calls are invoked to perform specific behaviors. For example, the API call `getLine1Number()` is used to return the phone number of the device, which can be matched with the "get—> phone number" behavior. However, expert knowledge is needed to link the meanings of line1 number with the phone number. Similar to permissions and intents, we also leverage the descriptions of API calls to help us match them with given sensitive behaviors. Table VII lists an example of API call matching.

Unfortunately, not all the identified sensitive behaviors have corresponding successfully matched API calls. For example, Fig. 2 presents the snippets of deleting a text message, which is implemented by using the `ContentResolver:delete()` function with the argument of parsed string `content://sms/conversations/`. Directly matching the given sensitive behavior with the invoked API call in the method body is not sound for such cases.

To address this problem, the method body is initially tokenized into a bag of words. For example, `ContentResolver:delete()` is tokenized as {"content," "resolver," "delete"}. Then, we check whether each word in the sensitive behavior is contained in the word bag. In this way, the snippets in Fig. 2 are matched with "delete—> text message" sensitive behavior based on the matched words "delete" and "sms" (similar meaning as "text message") that are marked in red.

Next, to perform malware analysis with machine learning algorithms, each app should be represented as a feature vector. Specifically, for a set of n given apps $X = \{x_1, x_2, \ldots, x_n\}$ and a set of k identified sensitive behavior feature $F = \{f_1, f_2, \ldots, f_k\}$, each app x_i is represented as a feature vector $\mathbf{x_i} = \langle x_{i1}, x_{i2}, \ldots, x_{ik}\rangle$, where x_{ij} denotes the value of the jth feature for the ith app. x_{ij} is calculated based on the above two matching rules in Algorithm 2.

Algorithm 2: Calculation of Feature Value.

Input:
x_i, f_j // x_i denotes the i^{th} app and f_j denotes the j^{th} sensitive behavior feature.

Output:
x_{ij} // x_{ij} denotes the output feature value.

1: $x_{ij} = 0$
2: $Per_{x_i} = \{per\}$ // Per_{x_i} denotes the required permission set of x_i.
3: **if** $\exists per \in Per_{x_i}$ and $MatchingRule - I(per, f_j)$ **then**
4:     $x_{ij}++$
5: **end if**
6: $Int_{x_i} = \{int\}$ // Int_{x_i} denotes the used intent set of x_i.
7: **if** $\exists int \in Int_{x_i}$ and $MatchingRule - I(int, f_j)$ **then**
8:     $x_{ij}++$
9: **end if**
10: $Method_{x_i} = \{md\}$ // $Method_{x_i}$ denotes the method set of x_i.
11: **for** each md in $Method_{x_i}$ **do**
12:     $API_{md} = \{api\}$ // API_{md} denotes the API call set of md.
13:     **if** $\exists api \in API_{md}$ and $MatchingRule - II(api, f_j)$ **then**
14:         $x_{ij}++$
15:     **else**
16:         $WdBag_{md} = \{wd\}$ // $WdBag_{md}$ denotes the word bag of md.
17:         **if** $MatchingRule - II(WdBag_{md}, f_j)$ **then**
18:             $x_{ij}++$
19:         **end if**
20:     **end if**
21: **end for**
22: **return** x_{ij}

In Algorithm 2, x_{ij} is initially set as 0 (line 1). Then, we extract a required permission set Per_{x_i} and an intent set Int_{x_i} from the `AndroidManifest.xml` file of app x_i. After that, we match each permission and intent in the two sets with the given sensitive behavior f_j with matching rule I, and increase the feature value with 1 if there exists a successful matching (lines 2–9). Next, we construct a method set $Method_{x_i}$ by extracting the methods from the Dalvik code, and match each method with f_j with matching rule II (lines 10–21). Note that our features are different from the binary features (e.g., permissions and API calls) that are set as 1 or 0, we not only consider the occurrence of corresponding sensitive behavior but also calculate its frequency of occurrence. By doing so, the feature vector constructed for each app contains more information than those constructed based on binary features.

D. MD and Classification

Finally, we conduct two malware analysis tasks, MD, and FC. Note that the labels attached to the feature vectors for the two tasks are different.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

For the task of MD, there are two types of labels, malicious, and benign, which are denoted as 1 and 0, respectively. In other words, if a given app x_i is a malicious one, then its corresponding label y_i is set as 1, or the label is set as 0 if the app is benign. However, for the task of malware classification, the label y_i belongs to one of the malware family names, such as *geinimi* or *droidkungfu*.

Therefore, for each task a dataset is initially constructed and represented as $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. Then, the dataset is split into a training dataset and a testing dataset. By applying known machine learning algorithms on the training dataset, different classifiers are generated. After that, each sample x_i in the testing dataset will be fed into the classifier and a label y'_i will be returned. If y'_i is equal to y_i, then the sample is correctly classified with the generated classifier, or it is wrong.

III. Evaluation

To evaluate CTDroid, we first introduce the study setup of our experiments, and then address the following five research questions.

RQ 1: Which classifier and parameters (i.e., θ and ϵ) are appropriate for CTDroid? (Section III-B)

RQ 2: Can CTDroid detect Android malware with high TPR and low FPR? (Section III-C)

RQ 3: Can CTDroid classify Android malware into their correct families with high accuracy? (Section III-D)

RQ 4: Can CTDroid handle a large scale of apps with high efficiency? (Section III-E)

RQ 5: To what extent is CTDroid resistant to code obfuscation techniques? (Section III-F)

A. Study Setup

1) Data Collection: CTDroid analyzes two main types of datasets: First, technical blogs, which contain the natural language contents about Android malware. Second, Android malware and benign apps, which are used to evaluate the performance of CTDroid for MD and FC.

a) Technical Blogs: In general, the technical blogs are written by researchers with specialized knowledge. Therefore, we utilize the contents in the technical blogs to mine sensitive behaviors that might do harmful activities to users potentially. The corpus of technical blogs is crawled from ten websites, including nine security companies websites [32]–[40] and the well-known personal website of Jiang [41], from 2010 to 2017. Given that we focus on Android malware analysis, we use the keywords such as "Android," "malware," and "malicious" to filter out the irrelevant blogs. We pick these ten security websites because of their expert analysis on Android malware, and we believe in their analysis result described in the crawled technical blogs. In summary, we collect 1385 Android malware related technical blogs that are listed in Table VIII. The time distribution of the collected blogs is illustrated in Fig. 3. The collected blogs as well as their extracted behaviors can be found online.[3]

[3][Online]. Available: https://drive.google.com/file/d/1slhs9kCm4leQ2S01SP6MR3vsId4VXcnG/view?usp=sharing

TABLE VIII
DESCRIPTIONS OF COLLECTED TECHNICAL BLOGS

Website	# Blogs	Website	# Blogs
Fortinet [32]	103	Lookout [37]	145
Secure List [33]	179	Cheetah Mobile [38]	50
Security Intelligence [34]	142	Palo Alto [39]	39
Trend Micro [35]	220	Check Point [40]	155
McAfee [36]	324	Jiang [41]	28

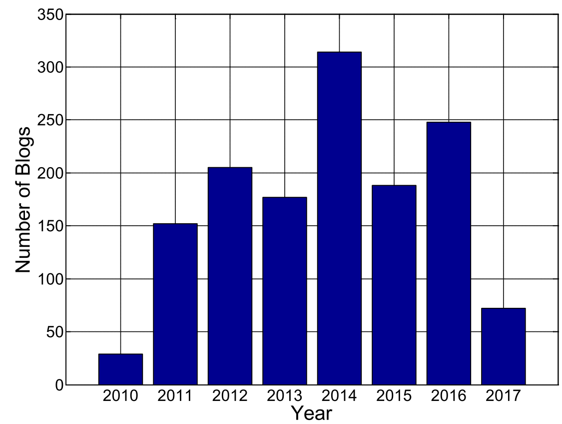


Fig. 3. Time distribution of collected technical blogs.

TABLE IX
DESCRIPTIONS OF THREE DATASETS USED FOR MD

Dataset	#Malware	#Benign Apps
MD-I	1,260	1,260
MD-II	5,560	5,560
MD-III	8,407	8,407
MD-IV	1,015	1,015

b) Android Malware and Benign Apps: To evaluate the performance of CTDroid for MD and FC, we apply it on four malware datasets, including three widely used datasets provided by Gnome project [42], Drebin [3], and FalDroid [43], and a new dataset constructed by ourselves by collecting recent malware samples from Palo Alto [39]. Specifically, for MD, we collect an equal number of most popular (10 000+ downloads) benign apps from Google Play [44] in the same period and add them to the four provided malware datasets. Each benign app has been uploaded to the VirusTotal [45], a website that contains 50+ virus engines, to make sure that no virus engine reports it as malicious. Therefore, four datasets that contain both malware and benign apps are constructed for MD. For convenience, the four datasets are named as MD-I, MD-II, MD-III, and MD-IV, and their descriptions are listed in Table IX. For FC, given that we need to split each dataset into a training set and a testing set, we remove the malware families that contain only one sample. For convenience, the four datasets are named as FC-I, FC-II, FC-III, and FC-IV, and their descriptions are listed in Table X.

2) Evaluation Metrics: For MD, the TPR is used to denote the percentage of malware that are correctly predicted as malware, and the FPR is used to denote the percentage of benign apps that are incorrectly predicted as malware. The goal of any MD research is to achieve a high value for TPR and a low value for FPR. For FC, the term classification accuracy is used to

TABLE X
DESCRIPTIONS OF THREE DATASETS USED FOR FC

Dataset	#Malware	#Families
FC-I	1,247	33
FC-II	5,513	132
FC-III	8,407	36
FC-IV	1,015	69

denote the percentage of malware that are correctly classified into their corresponding families.

3) Baseline Approaches: We compare the performance of CTDroid in MD and FC with three baseline approaches, i.e., FeatureSmith [30], FalDroid [43], and MaMaDroid [46]. The descriptions of the three baseline approaches are listed as follows.

1) Zhu and Dumitras proposed FeatureSmith [30], which first identifies 173 concrete features, including permissions, API calls, and intents, which occur in scientific papers. Then, they extract the identified features from the `AndroidManifest.xml` file and Dalvik code for MD. There are three main differences between our work and FeatureSmith. We will discuss them in Section V-C.
2) Fan *et al.* proposed FalDroid [43], which first constructs fregraphs from the malware samples within the same family to denote their common malicious behaviors. Then, they regard each fregraph as a feature and construct a feature space for malware analysis.
3) Mariconti *et al.* proposed MaMaDroid [46], which first builds a behavioral model in the forms of a Markov chain from the sequence of extracted API calls performed by apps. Then, it extracts features from the Markov chain to perform malware analysis.

All the experiments are conducted with a ten-fold cross validation on a quad-core 3.20 GHz PC running Ubuntu 14.04(64 bit) with 16 GB RAM and 1 TB hard disk.

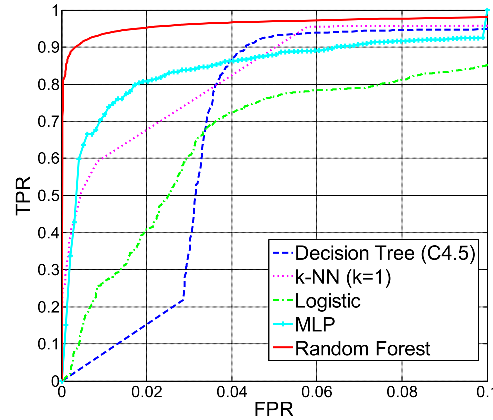


Fig. 4. Detection performance of CTDroid on the combined dataset with five different classifiers.

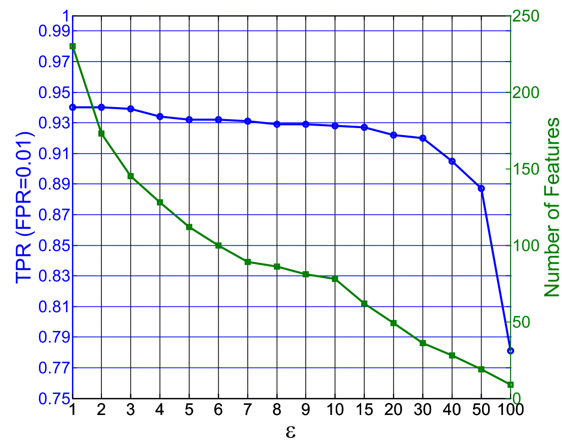


Fig. 5. TPR values (FPR $= 0.01$) of CTDroid on the combined dataset and numbers of generated sensitive behavior features with different ϵ.

*B. **RQ** 1: Which Classifier and Parameters (i.e., θ and ϵ) are Appropriate for CTDroid?*

To choose the appropriate classifier for CTDroid, five different machine learning algorithms, including decision tree [47], k-nearest neighbors [48], logistic [49], multilayer perceptron [50], and random forest [51], are applied in our approach. Specifically, we first combine the four datasets and remove the same samples. The combined dataset contains 11535 distinct samples. We also add the same number of benign apps into the combined dataset. Then, we construct five corresponding classifiers based on these algorithms and apply them for MD on the combined dataset. Note that here we initially set our two important parameters, i.e., θ and ϵ, as 0.9 and 3, respectively.

Fig. 4 illustrates the MD performance of CTDroid on the combined dataset with five different classifiers. The result shows that random forest outperforms the other four classifiers. When FPR is 0.01, the TPR of random forest can achieve 0.939, much higher than those of the other classifiers. The main reason for the superior performance of random forest is that it is an ensemble classifier that leverages the out-of-bag errors as an estimate of the generalization error to improve its performance, whereas the others are base classifiers. Therefore, due to the superior performance of random forest among the five classifiers, random forest is selected as our default classifier in later experiments.

Next, we investigate the influence of θ and ϵ to our performance. θ controls the similarity calculation between extracted behaviors; ϵ controls the threshold value of filtering out useless clusters. To set an appropriate θ, we manually construct a set of behaviors with similar meanings and then calculate their similarities between any two behaviors. We find that all the calculated similarities are higher than 0.9. Thus, θ is set as 0.9 in this paper. ϵ is a parameter to balance the size of feature space and detection performance. The higher of ϵ, the fewer sensitive behavior features will be, but we might miss some significant features if ϵ is too high. However, if the ϵ is too low, we might introduce some useless features. To select an appropriate ϵ, we vary the values of ϵ as $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 100\}$. The TPR values (FPR $= 0.01$) of CTDroid and numbers of generated sensitive behavior features with different ϵ are illustrated in Fig. 5. We find that with the increase of ϵ, the number of features decreases. Moreover, the TPR value starts to decrease when ϵ is higher than 3. Therefore, to achieve high performance, ϵ is set as 3.

When the θ and ϵ are set as 0.9 and 3, respectively, in this paper, 145 features are extracted from collected blogs. For

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

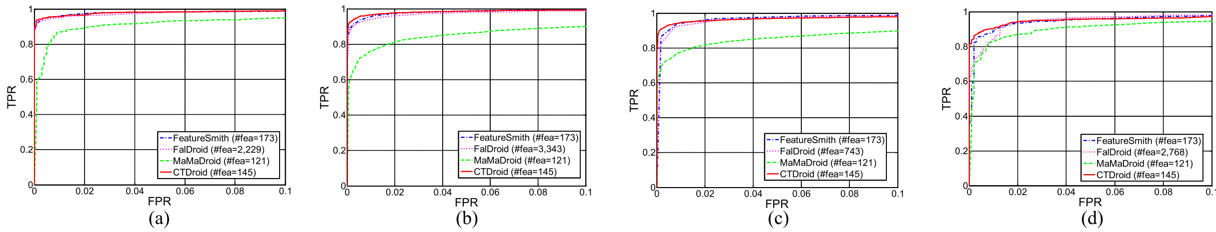


Fig. 6. MD performance of CTDroid and three baseline approaches with all features on four MD datasets. (a) MD-I dataset. (b) MD-II dataset. (c) MD-III dataset. (d) MD-IV dataset.

the extracted features, we find that the most frequent behavior feature is "send—> text message." The following features are about getting personal information, including the phone number, device id, imei, and the serial number. For the verbs of extracted features, "get," "send," and "use" are the most common ones, indicating that getting personal information and sending them to the remote server are the prevailing malicious behaviors for existing malware samples. Moreover, we find that there exists one interesting behavior feature, i.e., "make—> screenshot," which is not used in existing methods. The screenshot will be stored and send to the remote server instead of the exact personal information, thus making it hard to detect with only permission or API features.

Answer to RQ 1: We select random forest as our default classifier due to its superior performance. In addition, θ and ϵ are set as 0.9 and 3, respectively.

C. ***RQ*** *2: Can CTDroid Detect Android Malware With High TPR and Low FPR?*

To answer RQ 2, we evaluate the MD performance of CTDroid on four datasets and compare it with three baseline approaches, i.e., FeatureSmith [30], FalDroid [43], and MaMaDroid [46]. Specifically, for each dataset, we train four random forest classifiers but four different feature sets. In this paper, when $\epsilon = 3$, 145 features are extracted from collected blogs. For FeatureSmith, 173 features are identified from scientific papers. However, for FalDroid, its feature space is constructed from the training dataset, thus the feature number of FalDroid varies from different datasets. For MaMaDroid, 121 features are extracted.

Fig. 6 presents the MD performance of CTDroid and three baseline approaches with their all features using receiver operating characteristic plots. The plots illustrate the relationship between the TPRs and the FPRs of the four classifiers. Note that the term #fea in Fig. 6 denotes the number of extracted features. The results demonstrate that CTDroid gets an almost equally performance with FeatureSmith and FalDroid, and performs better than MaMaDroid on the four datasets. For example, on the MD-I dataset, when the FPR is 0.01, all of CTDroid, FeatureSmith, and FalDroid have a TPR around 0.958, while the TPR of MaMaDroid is only 0.869.

Even CTDroid gets a fairly good performance for MD when FPR is 0.01, it still incorrectly classifies about 13 benign apps as malware on the MD-I dataset. The main reason to explain the incorrectly classified samples is that some words contain multimeanings, thus affecting the performance of semantic matching approach introduced in Section II-C. For example, if a method body contains the string value "Please contact me," CTDroid would extract the wrong sensitive concept "contact" and incorrectly identify the related features from the method.

Recall that our proposed features not only consider the occurrence of corresponding sensitive behavior, but also calculate its frequency of occurrence, thus our features would contain more information than the binary features generated by FeatureSmith and FalDroid. To evaluate the effectiveness with few features, we compare the detection performance of CTDroid and baseline approaches with top-m features ranked by information gain [52].

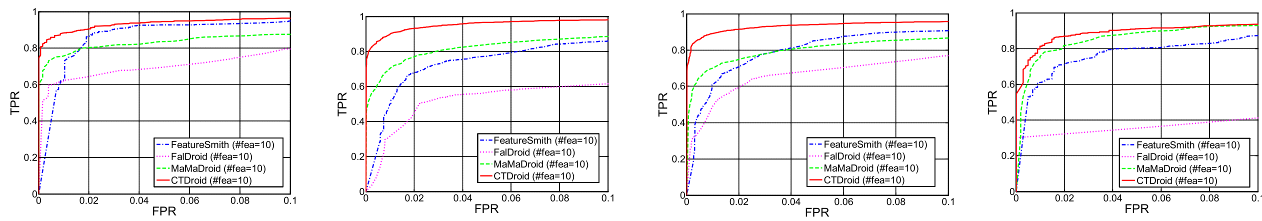
Fig. 7 presents the detection performance of the four approaches with top ten ($m = 10$) features ranked by information gain. The results on all the four datasets demonstrate that with only ten features CTDroid outperforms the baseline approaches due to the more informative features. For example, on the MD-II dataset, when the FPR is 0.01, CTDroid gets a TPR of 0.9, while the TPRs of FeatureSmith, FalDroid, and MaMadroid are 0.503, 0.340, and 0.710, respectively.

To further investigate the information gain of different features, we vary the values of m from 1 to 50 and calculate the cumulative information gain of the top-m ranked features. As illustrated in Fig. 8, on all the four datasets, the cumulative information gains of CTDroid are higher than those of baseline approaches. When m is set as 10, the cumulative information gain of CTDroid is 1.243 on the MD-II dataset, more than those of FeatureSmith, FalDroid, MaMaDroid, i.e., 1.130, 0.978, 1.144. Moreover, we find that the cumulative information gain of MaMaDroid hardly changes when m is higher than about 15, indicating that most of the features extracted by MaMaDroid have little significance for malware analysis.

Answer to RQ 2: CTDroid gets a high performance for MD with a TPR of 0.958 when the FPR is only 0.01. Furthermore, the features extracted by CTDroid are more informative than those of three baseline approaches.

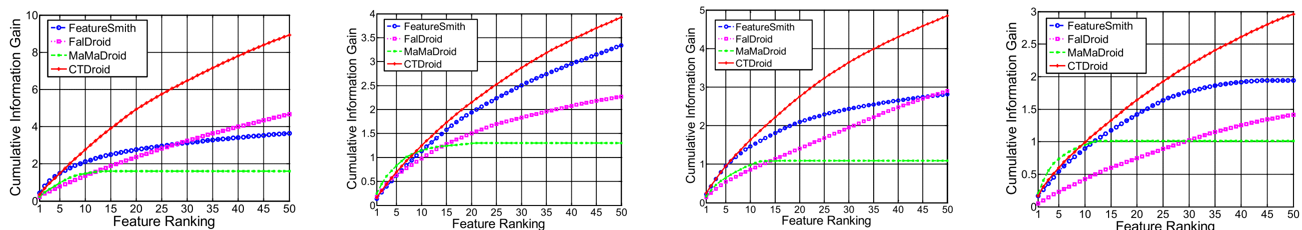
D. ***RQ*** *3: Can CTDroid Classify Android Malware Into Their Correct Families With High Accuracy?*

To evaluate the FC performance of CTDroid, we apply it on the four datasets, i.e., FC-I, FC-II, FC-III, and FC-IV. Furthermore, we compare CTDroid with the three baseline approaches. The results are listed in Table XI, where the values marked



(a) (b) (c) (d)

Fig. 7. MD performance of CTDroid and three baseline approaches with top ten features ranked by information gain on four MD datasets. (a) MD-I dataset. (b) MD-II dataset. (c) MD-III dataset. (d) MD-IV dataset.

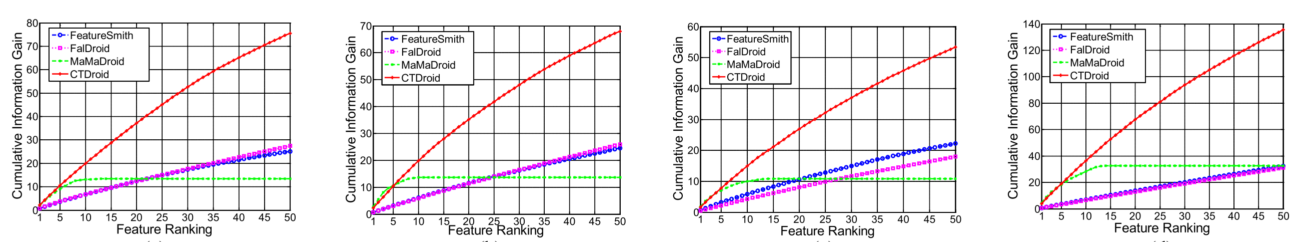


(a) (b) (c) (d)

Fig. 8. Cumulative information gain of features ranked by information gain on four MD datasets. (a) MD-I dataset. (b) MD-II dataset. (c) MD-III dataset. (d) MD-IV dataset.

TABLE XI
CLASSIFICATION PERFORMANCE OF CTDROID AND THREE BASELINE APPROACHES ON FOUR DIFFERENT DATASETS

Dataset	Approach	#Fea	Accuracy	#Fea	Accuracy
FC-I	FeatureSmith	173	0.940	10	0.868
	FalDroid	2,229	0.972	10	0.740
	MaMaDroid	121	0.860	10	0.859
	CTDroid	145	**0.979**	10	**0.935**
FC-II	FeatureSmith	173	0.950	10	0.674
	FalDroid	3,343	0.953	10	0.623
	MaMaDroid	121	0.878	10	0.874
	CTDroid	145	**0.961**	10	**0.911**
FC-III	FeatureSmith	173	0.918	10	0.662
	FalDroid	743	**0.942**	10	0.592
	MaMaDroid	121	0.849	10	0.838
	CTDroid	145	0.922	10	**0.859**
FC-IV	FeatureSmith	173	0.883	10	0.606
	FalDroid	2,768	0.876	10	0.581
	MaMaDroid	121	0.839	10	0.833
	CTDroid	145	**0.890**	10	**0.848**



(a) (b) (c) (d)

Fig. 9. Cumulative information gain of features ranked by information gain on four FC datasets. (a) FC-I dataset. (b) FC-II dataset. (c) FC-III dataset. (d) FC-IV dataset.

in bold denote the highest classification accuracy for each dataset.

In Table XI, columns 3 and 4 list the classification performance of the three approaches with their all features. With all the 145 sensitive behavior features, CTDroid performs best on FC-I dataset, FC-II dataset, and FC-IV dataset. However, on FC-III dataset, the accuracy of CTDroid is about 0.02 less than that of FalDroid. Columns 5 and 6 list the classification performance of the three approaches with top ten features ranked by information gain. With only ten features, CTDroid outperforms the

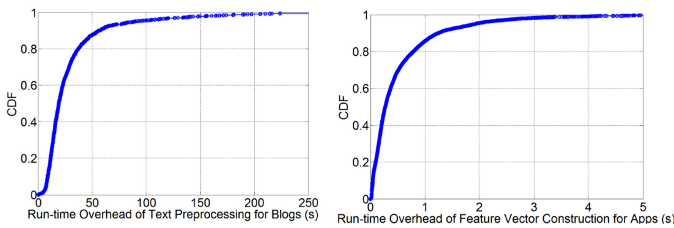


Fig. 10. CDFs of the run-time overhead for text preprocessing and feature vector construction.

baseline approaches. For example, on FC-III dataset, CTDroid can get an accuracy of 0.859, much higher than those of baseline approaches.

We further calculate the cumulative information gain of the three approaches when m varies from 1 to 50. As illustrated in Fig. 9, the results suggest that the cumulative information gains of CTDroid on all the four datasets are much higher than those of baseline approaches, indicating that our proposed features are more informative compared with those of baseline approaches.

Answer to RQ 3: CTDroid performs well on malware classification with a 97.9% accuracy. Furthermore, its accuracy can achieve 93.5% with only ten features, which is much better than three baseline approaches.

E. **RQ** 4: Can CTDroid Handle a Large Scale of Apps With High Efficiency?

To answer RQ 4, we investigate the run-time overhead of CTDroid. For the four procedures introduced in Section II, text preprocessing for blogs and feature vector construction for apps are the two main procedures that require more computation resource than the other two procedures. The cost of text preprocessing and feature construction depends on the number of collected blogs and the number of apps, respectively.

The cumulative distribution function of run-time overhead for the two procedures are illustrated in Fig. 10. The left figure presents that 91.3% blogs require less than 60 s to extract the behaviors from their content by using *Stanford Parser*. In total, 11 h is required to implement the text preprocessing for all the 1385 collected blogs. The right figure presents that 0.5 s is needed on average to construct a feature vector for each given app after the disassembling. The cost of disassembling of apks is the same as the other approaches, since it is the necessary step to analyze apps for all the three approaches statically. It is worth noting that the text preprocessing and the feature vector construction procedures could be conducted on several PCs in parallel, thus further reducing the total run-time overhead.

For the sensitive behavior generation procedure, with a set of about 208 K extracted behaviors as input, 10 min is needed for the clustering of behaviors and outputting the set of sensitive behaviors. Finally, the construction of random forest classifier used for malware analysis requires less than 1 min.

We also investigate the efficiencies of the three baseline approaches. For FeatureSmith, its run-time overhead is similar to ours, since both of the two approaches process feature engineering from contents written in natural language. However, FalDroid relies on graph analysis that requires about one week to construct the fregraph-based feature space. Furthermore, more than 2 s is needed to generate the feature vector for a given app after the disassembling. For MaMaDroid, about two days are needed to construct all the call graphs and extract feature vectors for all the apps.

Answer to RQ 4: CTDroid requires only 0.5 s on average to construct the feature vector for each app after its disassembling. The low run-time overhead allows CTDroid to work efficiently and be scalable to a large number of apps.

F. **RQ** 5: To What Extent is CTDroid Resistant to Code Obfuscation Techniques?

To evaluate the resilience of CTDroid to code obfuscation techniques, we only consider the techniques that try to increase the values of sensitive behavior features, since the technique of deleting code that reduces the feature values might affect the functionalities of original apps. For example, the code obfuscation techniques can add the value of feature "send—> text message" from 0 to 1, but it is hard to reduce it from 1 to 0 without affecting the app's functionality of sending messages.

In general, the code obfuscation techniques for Android malware can be categorized into two main groups: First, typical obfuscation techniques such as class renaming, inserting of useless instructions. Second, advanced obfuscation techniques such as reflection techniques and encryption packer.

For the typical obfuscation techniques, we first leverage the popular Android obfuscator named as *DashO* [53] to perform class renaming obfuscation techniques on 20 randomly selected samples. Then, we compare the feature vectors extracted from the original samples and the obfuscated samples. The results show that such typical obfuscation techniques would have no effect on our approach. However, the inserting of useless instructions might increase the feature values. For example, if the attacker inserts a string "we will send a text message" into a method, our approach will incorrectly match the "send—> text message" feature. This technique might misguide CTDroid to classify a benign app as malicious, but can hardly misguide a malware into benign. To evaluate the resilience of CTDroid to such technique, we first randomly select 100 malware from the MD-I dataset as the testing set. Then, we increase the values of t randomly selected features with 1. After that, we fed these obfuscated feature vectors into constructed classifier to detect whether their corresponding output labels are still 1, indicating that the obfuscation techniques do not affect the detection result. We vary the t from 1 to 145 and repeat this experiment 100 times. The results are shown in Fig. 11, where the false negative rate (FNR) denotes the percent of malware samples that are incorrectly classified as benign after the changing of feature vectors. We observe that when t is less than 20, nearly no

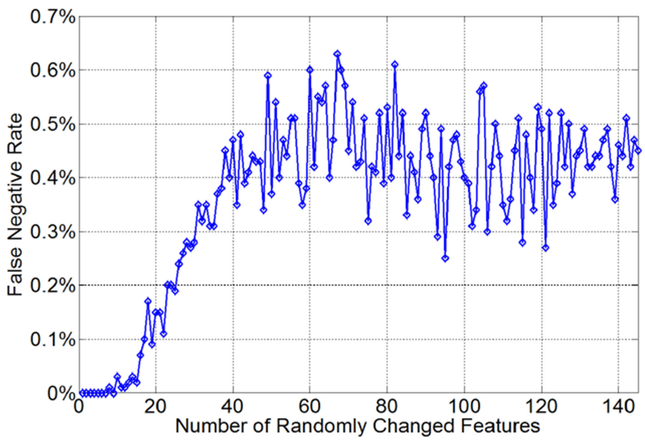


Fig. 11. FNRs of CTDroid for the obfuscation of increasing feature values by 1.

TABLE XII
FNRs of CTDroid for the Obfuscation of Increasing Feature Values From 1 to 5

Increased Feature Value	FNR
1	0.12%
2	0.28%
3	0.35%
4	0.38%
5	0.6%

malware is incorrectly classified. The highest FNR is 0.63%, indicating that on average less than 1 malware sample in the testing set is affected by the obfuscation techniques.

Moreover, we also vary the increased feature values as {1, 2, 3, 4, 5}. Note that here we fix the number of t as 20. Table XII lists the average FNRs with different increased feature values from 1 to 5. The results demonstrate that with the increase of feature values, the FNR increases. The main reason is that if the feature value changes a lot, the corresponding sample will be regarded as an abnormal one, thus causing the increase of FNR. To limit the affect caused by inserting useless instructions, it is a promising way to combine dynamic analysis techniques with our approach to filter out the code that would never be executed.

For the advanced typical techniques, the reflection techniques that can hide some function invocations would not affect the constructed feature vectors, since the API calls that use the reflection techniques are still contained in their method bodies. Specifically, to evaluate the resilience of CTDroid to reflection techniques, we randomly select 20 samples that use the reflection techniques. Then, we leverage *DroidRA* [54], an open-source tool, to perform reflection analysis on these samples to transform the reflection methods into normal methods. For example, the reflection method *getDeclaredMethod("getITelephony," null)* will be transformed to *getITelephony()*. The results show that the vectors have no changing. Therefore, our approach is resilient to the reflection techniques.

The encryption packer such as *Ijiami* [55] and *Bangcle* [56], can hide the actual Dex code, thus making the disassembled tools such as *apktool* [26] unable to obtain the Dalvik code. However, with existing unpacker tools such as *PackerGrind* [57] we can recover the actual Dex files.

As to the native code, since we limit our analysis to the Dalvik code, thus CTDroid might miss the malicious activities implemented in the native code. However, we could take advantages of existing binary analysis frameworks, such as *Angr* [58], which can help us analyze the native code and detect the malicious activities. We will explore this tool in future work.

Answer to RQ 5: CTDroid is resilient to typical obfuscation techniques and reflection techniques. In addition, CTDroid can handle advanced packing techniques by leveraging existing tools.

IV. Limitations and Threats to Validity

Our evaluation is subject to threats to validity, many of which are induced by limitations of our approach. The most important threats and limitations are listed as follows.

A. Threats to Internal Validity

In the sensitive behavior generation procedure, a set of Android system related concepts is used to filter out the useless behaviors. However, we cannot ensure the completeness of this set. Missing related concepts would make CTDroid miss sensitive behaviors. In future work, we plan to add more related concepts into this set by manually analyzing the malicious activities of recent malware samples.

B. Threats to External Validity

We extract the sensitive behavior features from a set of technical blogs collected from 2011 to 2017. Even these features work well on three widely used datasets, they might not be effective for the malware samples that are developed after 2017. In future work, we plan to collect more recent technical blogs and try to extract more detailed features for better malware analysis.

C. Matching of Abstract Behaviors

In addition to the specific sensitive behaviors generated by using 14 semantic groups and a set of Android system related concepts, there are some abstract behaviors that we fail to accurately match them with the Dalvik code. For example, we cannot detect whether a given app contains the abstract behavior "launch—> root exploits," since the presence of root exploits in malware relies on expected runtime environment (e.g., specific vulnerable device driver or preconditions) [59]. In future work, we plan to transform the abstract behaviors into a list of specific behaviors and then design more specific features to address the limitation of matching abstract behaviors.

V. Related Work

A. Android Malware Analysis

With the recent surge in research interest in the area of Android device security, a large number studies focusing on mobile malware analysis have been conducted. These studies fall into two general categories: signature-based and machine-learning-based approaches.

Signature-based approaches look for specific patterns of malware behavior. Enck *et al.* [60] proposed the Kirin security service for Android, which designs nine rule templates to match the undesirable properties in security configuration bundled with apps. Grace *et al.* [61] proposed a proactive scheme to spot zero-day Android malware, and developed a system called RiskRanker to analyze whether an app exhibits malicious behavior. Zhou *et al.* [62] proposed a permission-based behavioral footprinting scheme to detect new samples of known malware families and then applied a heuristic-based filtering scheme to identify inherent behaviors of unknown malware families. Feng *et al.* [63] proposed Astroid, which automatically generate the malware signature by analyzing a maximally suspicious common subgraph that is shared between all known instances of a malware family.

Machine learning-based approaches extract features from app code and apply standard machine learning algorithms to perform a classification task. Wang *et al.* [2] proposed an approach for malware analysis based on requested permissions, which are security-aware features that restrict the access of apps to the core facilities of devices. Arp *et al.* [3] proposed Drebin, which performs a broad static analysis, gathering as many features of an app as possible such as permissions, API calls, and strings in Dalvik code. These features are then embedded in a joint vector space for Android malware analysis. Meng *et al.* [64] proposed a precise semantic model of Android malware based on deterministic symbolic automaton, from which semantic features are extracted for malware analysis. Hou *et al.* [65] proposed HinDroid, which first constructs a structured heterogeneous information network and then uses multikernel learning to perform malware analysis.

B. NLP for Android

With the development of NLP techniques, there are some approaches that analyze the Android-related contextual content to improve the analysis of relative tasks, such as risk assessment, privacy analysis, and malware analysis.

Rahul *et al.* [66] proposed WHYPER, which leverages NLP and automates risk assessment of mobile apps by revealing discrepancies between app descriptions and their true functionalities. Qu *et al.* [67] proposed AutoCog, which can automatically assess description-to-permission fidelity of apps by extracting semantic information from the descriptions. Yu *et al.* [68] proposed PPChecker, which adopts NLP and program analysis techniques to automatically identify the incomplete, incorrect or inconsistent privacy policies. Slavin *et al.* [69] proposed a framework that detects the privacy violation based on a privacy-policy-phrase ontology and a set of matching from API calls to policy phrases. Gorla *et al.* [70] proposed CHABADA, which first groups the Android apps into clusters according to their description topics and then identify outliers in each cluster with respect to the API call usage.

C. Differences With FeatureSmith

The most related work is FeatureSmith proposed by Zhu and Dumitras [30]. FeatureSmith identifies 173 concrete named entities (i.e., permissions, API calls, and intents) that are associated with keywords (i.e., malware family names) in scientific papers with NLP techniques. Then, they apply these features for malware analysis.

There are three main differences between FeatureSmith and our work, which are as follows.

1) Different datasets used for feature engineering: FeatureSmith mines behaviors from scientific papers, while we mine the behaviors from the technical blogs. We believe that the technical blogs contain more detailed descriptions about Android malware behaviors rather than scientific papers due to their page limit.
2) Different sensitive behavior identifying methods: FeatureSmith identifies the sensitive behaviors with the keywords of malware families, such as *geinimi* and *droidkungfu*. However, this method might lose some behaviors that do not occur with such keywords. To address this limitation, we propose a clustering-based approach to mine the sensitive behaviors among all the extracted behaviors.
3) Different sensitive behavior matching rules: FeatureSmith matches the behaviors to concrete features based on keyword search in the contextual content. Thus, they cannot handle the content that does not contain any concrete features, while we propose two matching rules to bridge the sematic gap between the programming language with the sensitive behaviors.

VI. Conclusion

In this paper, we proposed a novel system, CTDroid, to automatically construct informative features for malware analysis by analyzing a corpus of Android malware related technical blogs. To evaluate the effectiveness of constructed features, we evaluated CTDroid for two tasks, i.e., MD and FC. Our extensive evaluation results showed that CTDroid can achieve high accuracy and efficiency. Furthermore, our features presented more information than those of binary features proposed by the state-of-the-art approaches.

References

[1] Qihoo Inc., "Report of smartphone security in China," 2017. [Online]. Available: http://zt.360.cn/1101061855.php?dtid=1101061451&did=491056914

[2] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in Android applications for malicious application detection," *IEEE Trans. Inf. Forensics Secur.*, vol. 9, no. 11, pp. 1869–1882, Nov. 2014.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.

[4] Y. Aafer, W. Du, and H. Yin, “Droidapiminer: Mining API-level features for robust malware detection in Android,” in *Proc. 9th Int. ICST Conf., SecureComm*, 2013, pp. 86–103.
[5] J. Garcia, M. Hammad, and S. Malek, “Lightweight, obfuscation-resilient detection and family identification of Android malware,” *ACM Trans. Softw. Eng. Methodology*, vol. 26, no. 3, 2018, Art. no. 11.
[6] J. Crussell, C. Gibler, and H. Chen, “Attack of the clones: Detecting cloned applications on Android markets,” in *Proc. Eur. Symp. Res. Comput. Secur.*, 2012, pp. 37–54.
[7] J. Crussell, C. Gibler, and H. Chen, “Andarwin: Scalable detection of semantically similar Android applications,” in *Proc. Eur. Symp. Res. Comput. Secur.*, 2013, pp. 182–199.
[8] K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on Android markets,” in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 175–186.
[9] N. Marastoni, A. Continella, D. Quarta, S. Zanero, and M. D. Preda, “Groupdroid: Automatically grouping mobile malware by extracting code similarities,” in *Proc. 7th Softw. Secur., Protection, Reverse Eng./Softw. Secur. Protection Workshop*, 2017, Paper 1.
[10] M. Fan *et al.*, “Frequent subgraph based familial classification of Android malware,” in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng.*, 2016, pp. 24–35.
[11] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, “Structural detection of Android malware using embedded call graphs,” in *Proc. ACM Workshop Artif. Intell. Secur.*, 2013, pp. 45–54.
[12] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, “DAPASA: Detecting Android piggybacked apps through sensitive subgraph analysis,” *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 8, pp. 1772–1785, Aug. 2017.
[13] M. Fan *et al.*, “Graph embedding based familial analysis of Android malware using unsupervised learning,” in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 771–782.
[14] “Security alert: New BeanBot SMS trojan discovered,” 2011. [Online]. Available: https://www.csc2.ncsu.edu/faculty/xjiang4/BeanBot/
[15] “jsoup: Java html parser,” 2018. [Online]. Available: https://jsoup.org/
[16] S. Schuster and C. D. Manning, “Enhanced English universal dependencies: An improved representation for natural language understanding tasks,” in *Proc. 10th Int. Conf. Lang. Resour. Eval.*, 2016, pp. 2371–2378.
[17] “The Stanford parser: A statistical parser,” 2018. [Online]. Available: https://nlp.stanford.edu/software/lex-parser.shtml
[18] “Dropping common terms: Stop words,” 2018. [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms%-stop-words-1.html
[19] “Wordnet: A lexical database for English,” 2018. [Online]. Available: https://wordnet.princeton.edu/wordnet/
[20] A. I. Anton and J. B. Earp, “A requirements taxonomy for reducing web site privacy vulnerabilities,” *Requirements Eng.*, vol. 9, no. 3, pp. 169–185, 2004.
[21] J. Ramos, “Using TF-IDF to determine word relevance in document queries,” in *Proc. 1st Instructional Conf. Mach. Learn.*, 2003, pp. 1–4.
[22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.
[23] “Wikipedia,” 2017. [Online]. Available: https://en.wikipedia.org/wiki/Main_Page
[24] D. Guthrie, B. Allison, W. Liu, L. Guthrie, and Y. Wilks, “A closer look at skip-gram modelling,” in *Proc. 5th Int. Conf. Lang. Resour. Eval.*, 2006, pp. 1222–1225.
[25] A. P. Felt, S. Egelman, and D. Wagner, “I’ve got 99 problems, but vibration ain’t one: A survey of smartphone users’ concerns,” in *Proc. 2nd ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2012, pp. 33–44.
[26] “Apktool: A tool for reverse engineering Android apk files,” 2018. [Online]. Available: https://ibotpeaches.github.io/Apktool//
[27] M. Li *et al.*, “LibD: Scalable and precise third-party library detection in Android markets,” in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 335–346.
[28] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, “An investigation into the use of common libraries in Android apps,” in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 403–414.
[29] 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.
[30] Z. Zhu and T. Dumitras, “Featuresmith: Automatically engineering features for malware detection by mining the security literature,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 767–778.
[31] “Android developer,” 2018. [Online]. Available: https://developer.android.com/index.html
[32] “Fortinet,” 2017. [Online]. Available: https://blog.fortinet.com
[33] “Securelist,” 2017. [Online]. Available: https://securelist.com
[34] “SecurityIntelligence,” 2017. [Online]. Available: https://securityintelligence.com
[35] “Trendlabs,” 2017. [Online]. Available: http://blog.trendmicro.com
[36] “McAfee,” 2017. [Online]. Available: https://securingtomorrow.mcafee.com
[37] “Lookout,” 2017. [Online]. Available: https://www.lookout.com/
[38] “Cheetah Mobile,” 2017. [Online]. Available: http://www.cmcm.com/blog/en/
[39] “Paloalto,” 2017. [Online]. Available: https://www.paloaltonetworks.com/
[40] “Checkpoint,” 2017. [Online]. Available: https://blog.checkpoint.com/
[41] “Mobile security alerts,” 2012. [Online]. Available: https://www.csc2.ncsu.edu/faculty/xjiang4/alerts.html/
[42] Y. Zhou and X. Jiang, “Dissecting Android malware: Characterization and evolution,” in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.
[43] M. Fan *et al.*, “Android malware familial classification and representative sample selection via frequent subgraph analysis,” *IEEE Trans. Inf. Forensics Secur.*, vol. 13, no. 8, Aug. pp. 1890–1905, 2018.
[44] “Google Play,” 2018. [Online]. Available: https://play.google.com/store
[45] “Virustotal: A free service that analyzes all kinds of malware,” 2017. [Online]. Available: https://www.virustotal.com/
[46] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, “Mamadroid: Detecting Android malware by building markov chains of behavioral models,” *ACM Trans. Privacy Secur.*, vol. 22, no. 2, p. 14, 2019.
[47] S. L. Salzberg, “C4.5: Programs for machine learning,” *Mach. Learn.*, vol. 16, no. 3, pp. 235–240, 1994.
[48] D. W. Aha, D. Kibler, and M. K. Albert, “Instance-based learning algorithms,” *Mach. Learn.*, vol. 6, no. 1, pp. 37–66, 1991.
[49] S. Le Cessie and J. C. Van Houwelingen, “Ridge estimators in logistic regression,” *Appl. Statist.*, vol. 41, no. 1, pp. 191–201, 1992.
[50] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, “The multilayer perceptron as an approximation to a Bayes optimal discriminant function,” *IEEE Trans. Neural Netw.*, vol. 1, no. 4, pp. 296–298, Dec. 1990.
[51] L. Breiman, “Random forests,” *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[52] L. E. Raileanu and K. Stoffel, “Theoretical comparison between the gini index and information gain criteria,” *Ann. Math. Artif. Intell.*, vol. 41, no. 1, pp. 77–93, 2004.
[53] “Dasho,” 2018. [Online]. Available: https://www.preemptive.com/products/dasho/overview
[54] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, “Droidra: Taming reflection to support whole-program analysis of Android apps,” in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 318–329.
[55] “Ijiami Inc.,” 2018. [Online]. Available: http://www.ijiami.cn/
[56] “Bangcle Inc.,” 2018. [Online]. Available: http://www.bangcle.com/, 2018.
[57] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, “Adaptive unpacking of Android apps,” in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 358–369.
[58] Y. Shoshitaishvili *et al.*, “SOK: (state of) the art of war: Offensive techniques in binary analysis,” in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 138–157.
[59] I. Gasparis, Z. Qian, C. Song, and S. V. Krishnamurthy, “Detecting Android root exploits by learning from root providers,” in *Proc. 26th USENIX Conf. Secur. Symp.*, 2017, pp. 1129–1144.
[60] W. Enck, M. Ongtang, and P. McDaniel, “On lightweight mobile phone application certification,” in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009.
[61] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “Riskranker: Scalable and accurate zero-day Android malware detection,” in *Proc. MobiSys*, 2012, pp. 235–245.
[62] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, “Viewdroid: Towards obfuscation-resilient mobile application repackaging detection,” in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 25–36.
[63] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, “Automated synthesis of semantic malware signatures using maximum satisfiability,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.
[64] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, “Semantic modelling of Android malware for effective malware comprehension, detection, and classification,” in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 306–317.
[65] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, “Hindroid: An intelligent Android malware detection system based on structured heterogeneous information network,” in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2017, pp. 1507—1515.

[66] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. 22nd USENIX Conf. Secur.*, 2013, pp. 527–542.

[67] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in Android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1354–1365.

[68] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of Android apps?" in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2016, pp. 538–549.

[69] R. Slavin *et al.*,"Toward a framework for detecting privacy policy violations in Android application code," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 25–36.

[70] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1025–1035.

Ming Fan received the B.S. and Ph.D. degrees in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 2013 and 2019, respectively, and the second Ph.D. degree in computing from The Hong Kong Polytechnic University, Hong Kong, in 2019.

He is currently an Assistant Professor with the School of Cyberspace Security, Xi'an Jiaotong University, Xi'an, China. His research interests include trustworthy software and Android malware detection, and familial identification.

Xiapu Luo received the Ph.D. degree in computer science from The Hong Kong Polytechnic University, Hong Kong, in 2007.

He was a Postdoctoral Research Fellow with the Georgia Institute of Technology, Atlanta, GA, USA. He is currently an Associate Professor with the Department of Computing and an Associate Researcher with the Shenzhen Research Institute, The Hong Kong Polytechnic University, Hong Kong. His research interests include smartphone security and privacy, network security and privacy, and Internet measurement.

Jun Liu received the B.S. and Ph.D. degrees in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 1995 and 2004, respectively.

He is currently a Professor with the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. He has authored more than 70 research papers in various journals and conference proceedings. His research interests include data mining and text mining.

Dr. Liu served as a Guest Editor for many technical journals, such as *Information Fusion*, IEEE SYSTEMS JOURNAL, and *Future Generation Computer Systems*. He also acted as a conference/workshop/track chair at numerous conferences.

Chunyin Nong received the B.S. degree in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 2017.

He is currently a Master with the Department of Computer Science and Technology, Xi'an Jiaotong University, China. His research interests include text mining and malware detection.

Qinghua Zheng received the B.S. degree in computer software, the M.S. degree in computer organization and architecture, and the Ph.D. degree in system engineering from Xi'an Jiaotong University, Xi'an, China, in 1990, 1993, and 1997, respectively.

He was a Postdoctoral Researcher with the Harvard University, Cambridge, MA, USA, in 2002. He is currently a Professor with the Department of Computer Science and Technology, Xi'an Jiaotong University. His research interests include knowledge engineering of big data and software trustworthiness evaluation.

Ting Liu received the B.S. and Ph.D. degrees in automation from Xi'an Jiaotong University, Xi'an, China, in 2003 and 2010, respectively.

He was a Visiting Professor with Cornell University, Ithaca, NY, USA. He is currently a Professor with the Systems Engineering Institute, Xi'an Jiaotong University. His research interests include software security and smart grids security.