This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

DAPASA: Detecting Android Piggybacked Apps through Sensitive Subgraph Analysis

Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian and Ting Liu[§]

***Abstract*—With the exponential growth of smartphone adoption, malware attacks on smartphones have resulted in serious threats to users, especially those on popular platforms, such as Android. Most Android malware are generated by piggybacking malicious payloads into benign applications (apps), which are called piggybacked apps. In this work, we propose DAPASA, an approach to detect Android piggybacked apps through sensitive subgraph analysis. Two assumptions are established to reflect different invocation patterns of sensitive APIs in the injected malicious payloads (rider) of a piggybacked app and in its host app (carrier). With these two assumptions, DAPASA generates a sensitive subgraph (SSG) to profile the most suspicious behavior of an app. Five features are constructed from SSG to depict the invocation patterns. The five features are fed into machine learning algorithms to detect whether the app is piggybacked or benign. DAPASA is evaluated on a large real-world dataset consisting of 2,551 piggybacked apps and 44,921 popular benign apps. Extensive evaluation results demonstrate that the proposed approach exhibits an impressive detection performance compared to that of three baseline approaches even with only five numeric features. Furthermore, the proposed approach can complement permission-based approaches and API-based approaches with the combination of our five features from a new perspective of the invocation structure.**

***Index Terms*—Piggybacked apps, Sensitive API, Sensitive subgraph, Malware detection, Static analysis**

I. Introduction

A. Background

ANDROID smartphones have recently gained much popularity. Many Android application (app) markets such as Google Play [1] and Anzhi Market [2] have been set up, where users can download various apps. The Android platform has become a major target of malware. As reported in a recent study conducted by Qihoo [3], about 37,000 Android malware attacks were detected daily in the first quarter of 2016.

According to Zhou *et al.* [4], piggybacking is one of the most common techniques utilized by malware authors to piggyback malicious payloads on popular apps to produce malware. About 86% of their collected 1,260 samples were piggybacked versions of legitimate apps with malicious payloads. The malware created through piggybacking is called a *piggybacked app* [5], [6]. A piggybacked app has two main parts, namely, the original benign code and the injected malicious payloads. Following the conventions described in [5], we use the term *carrier* to refer to the former and the term *rider* to refer to the latter.

Developing new malware from scratch is labor intensive, but malware authors can easily add a specific rider into various carriers through piggybacking techniques to quickly produce and distribute a large number of piggybacked apps [7]–[10]. For example, members of the notorious malware family *Geinimi* usually repackage themselves into various legitimate game apps, steal personal information and send it to a remote server. Typically, malware authors download paid apps from the official market, disassemble them, add malicious payloads, reassemble and submit the "new" apps to the official or alternative Android markets for free. The new piggybacked apps would entice smartphone users to download and install.

With the inclusion of the new rider code, piggybacked apps pose significant security threats, and effective techniques to detect them are necessary. Many research prototypes have been implemented to detect malware. The main challenge for current approaches is to fight against malware variants and zero-day malware. Current commercial anti-virus systems and signature-based approaches [11]–[14], which look for specific patterns in the bytecode, are effective in identifying known malware relying on signatures. However, they are easily evaded by bytecode-level transformation attacks [15]. Therefore, many other approaches [16]–[20] based on machine learning have been developed. These approaches extract features from app behaviors (e.g., permission request and API calls) and apply machine learning algorithms to perform binary classification. Although these machine learning approaches obtain high detection accuracy, they extract features solely from external symptoms and do not seek for an accurate and complete interpretation of app behavior [21].

B. Overview of the Proposed Approach

In this work, we detect Android piggybacked apps by utilizing the distinguishable invocation patterns of sensitive APIs between the rider and carrier. Sensitive APIs are governed by permissions for apps to access sensitive information (e.g., the user's phone number or location) or to perform sensitive tasks (e.g., change the WIFI state, send messages). It is worth noting

[§] The corresponding author.

Ming Fan, Jun Liu, and Ting Liu are with MOEKLINNS, Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (fanming.911025@stu.xjtu.edu.cn, {liukeen, tingliu}@mail.xjtu.edu.cn).

Wei Wang is with the Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, 3 Shangyuancun, Beijing 100044, China (wangwei1@bjtu.edu.cn).

Haifei Li is with the Department of Computer Science, Union University, USA (hli@uu.edu).

Zhenzhou Tian is with the School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, 710049, China (tianzhenzhou@xupt.edu.cn).

that sensitive APIs constitute only a small portion of the whole Android APIs and they cannot be easily obfuscated by existing techniques whereas the names of user-defined functions are usually obfuscated as *a*, *b* or *c*.

To further understand the distinguishable invocation patterns, we establish two assumptions based on an empirical analysis of piggybacked apps.

Assumption I: To perform its malicious task, the rider invokes more sensitive APIs than the carrier does.

Assumption II: Generally, in the rider, the cohesion of sensitive APIs, which is measured by calling distances, is higher than that in the carrier.

By exploiting the two assumptions, we develop DAPASA, an approach to detect Android piggybacked apps through sensitive subgraph analysis. DAPASA consists of the following four steps.

(1) DAPASA starts with the construction of a static function-call graph of a given app. It is a directed graph where nodes denote the functions invoked by the app and edges denote the actual calls among these functions.

(2) To differentiate the maliciousness of different sensitive APIs, DAPASA calculates the sensitivity coefficients for each sensitive API through a term frequency-inverse document frequency (TF-IDF)-like measure.

(3) DAPASA divides the static function-call graph into a set of subgraphs heuristically with sensitive API nodes and their nearby normal nodes. The subgraph with the highest sensitivity coefficient is selected as the sensitive subgraph to profile the most suspicious behavior of the given app.

(4) Five features are constructed from the sensitive subgraph. The feature sensitivity coefficient of the sensitive subgraph (scg) and the feature total sensitive distance of the sensitive subgraph (tsd) are used to measure the maliciousness and cohesion of sensitive APIs, respectively. In addition, three different types of sensitive motifs are exploited to further depict in a fine-grained manner the local invocation patterns of sensitive APIs. Finally, the five features are fed into machine learning algorithms to detect whether the app is piggybacked or benign.

DAPASA is evaluated on a large real-world dataset consisting of 2,551 piggybacked apps and 44,921 popular benign apps. The evaluation results show that DAPASA achieves good performance with a true positive rate of 95% and a false positive rate of 0.7%.

C. Contributions and Organization

The main contributions of this work are listed below.

(i) We propose two assumptions about the different invocation patterns of sensitive APIs between the rider and carrier in Android piggybacked apps. By exploiting these two assumptions, we construct a sensitive subgraph to represent the entire call graph and profile the most suspicious behavior of the given app.

(ii) We propose five numeric features from the generated sensitive subgraph. These features can not only be used for independent detection of piggybacked apps, but also have the ability to complement permission-based approaches and API-based approaches in the performance and explanation of the detection results.

(iii) We propose a TF-IDF-like measure to calculate the sensitivity coefficient of each sensitive API based on the idea of TF-IDF. It can reduce the interference factors of the sensitive APIs that frequently occur in both benign and malicious apps.

The remainder of this paper is organized as follows. Section II discusses related work. Section III introduces two important notations employed in this work. DAPASA is described in Section IV and evaluated in Section V. Section VI presents in-depth discussions. The conclusions and future work are followed in Section VII.

II. RELATED WORK

With the recent surge in research interest in the area of Android device security, a large number studies focusing on mobile malware detection have been conducted. The current work is related to three types of work described below.

A. Piggybacked App Detection

Several studies have provided approaches for piggybacked app detection; they can be categorized in two main groups, namely, static and dynamic analysis approaches.

Static analysis approaches investigate software properties by inspecting apps and their source code. The study most related to ours is the research conducted by Zhou and Jiang [4]. In their study, they classified the ways through which malware is installed in smartphones into three main categories: piggybacking, update attack, and drive-by download; piggybacking is the most common one. Zhou *et al.* proposed a fast and scalable approach called PiggyApp [5]. PiggyApp decouples the app code into primary and non-primary modules and extracts certain features, such as permissions and APIs used in the primary module, to detect piggybacked apps. Guan *et al.* [9] proposed a semantic-based approach called RepDetector, which first extracts input-output states of core functions in the app, and then compares the similarities between functions and apps. RepDetector is robust against obfuscation attacks relying on the semantic analysis of each app instead of syntax characteristics. Zhang *et al.* [8] proposed ViewDroid, which first designs a new birthmark (feature view graph) for Android apps based on the users navigation behavior across app views, and then calculates the similarity of the constructed birthmarks. ViewDroid is also robust against the obfuscation attacks since its high-level abstracted birthmark is not affected by low-level code obfuscation techniques. Both RepDetector and ViewDroid effectively detect app clones by finding similar pairs in the app market based on the proposed robust features.

Unlike static ones, dynamic analysis approaches typically extract run-time app information by instrumenting the Android framework. Lin *et al.* [22] proposed SCSdroid, which extracts the system call sequence of an app at run-time and uses the common subsequences to detect piggybacked apps in the same family. Isohara *et al.* [23] proposed kernel-based behavior

analysis, which generates a set of regular expression rules from the names and parameters of system calls of training malware, and detects the new apps by mapping their system calls and parameters with the rules. However, most dynamic analysis approaches require a representative set of execution paths and it is difficult to ensure that all the execution paths of the apps can be covered [24], [25].

Different from these studies, our approach focuses on the finding of different invocation patterns of sensitive APIs between the rider and the carrier instead of pair-wise apps similarity calculation in the market. The current work helps improve our understanding of the malicious behaviors of piggybacked apps.

B. Structure-based Analysis

The proposed approach is also related to approaches that are based on structure analysis of function-call graphs [26] or program dependence graphs [27]–[30].

Given that the identification of similarities in graphs is difficult, Gascon *et al.* [26] proposed an approach for malware detection based on efficient embedding of function-call graphs with an explicit feature map. Zhang *et al.* [21] implemented a prototype system called DroidSIFT, which classifies Android malware based on a weighted contextual API dependency graph. Chen *et al.* [31] proposed MassVet, which models the app's user interfaces as a directed graph, in which each node is a view within an app, and each edge describes the navigation (triggered by the input events) relations among the nodes. With similar view structures in different apps, MassVet can effectively identify piggybacked apps.

Several of these approaches analyze the entire graph, which might cause high overheads. In our work, only the sensitive API nodes and their nearby normal nodes are analyzed. Moreover, a sensitive subgraph is utilized to represent the entire function-call graph to reduce computational complexity.

C. Measurement of Features

An increasing number of features, such as permissions and APIs, are proposed for malware detection with machine learning algorithms. Different features have different contributions to the identification of maliciousness; therefore, measurement of features is significant.

For permission-based features, Moonsamy *et al.* [32] ranked the frequency of required and used permissions in malicious and benign datasets, respectively, to determine the most popular permission patterns requested by malicious and benign apps. Wang *et al.* [16] proposed three approaches, namely, mutual information, CorrCoef, and T-test, to rank permissions for improved understanding of their risk relevance. APIs are another widely used type of features for the detection of Android malware. Aafer *et al.* [33] identified the top APIs that invoked in Android malware, and analyzed the difference in usage between malware and benign apps. Suarez *et al.* [14] proposed an approach to measure how important a code block is to a malware family with TF-IDF.

Unlike these studies, the current work presents a TF-IDF-like approach to measure the sensitivity coefficient of each sensitive API by considering not only its frequency of occurrence in malicious and benign apps, but also its corresponding category information.

III. BASIC NOTATIONS

Two basic notations, static function-call graph and sensitive subgraph, are introduced in this section.

Given that limited resources impede monitoring apps at runtime, DAPASA performs a static analysis. It transforms the given app into a graph representation, namely, static function-call graph [26], [34], which contains the necessary structure information to profile the behaviors of an app.

Definition 1 Static Function-Call Graph (SFCG). SFCG is constructed with the call relations among the functions captured from Android apk files (the format of installation file of Android apps) with disassembling tools such as apktool [35]. It can be represented as a directed graph $SFCG = (V, E)$.

- $V = \{v_i | 1 \leq i \leq n\}$ denotes the set of functions invoked by a given app, in which each $v_i \in V$ corresponds to a function name.
- $E \subseteq V \times V$ denotes the set of function-calls, in which edge $(v_i, v_j) \in E$ indicates one call exists from caller function v_i to callee function v_j.

Intuitively, we provide the SFCG of a sample (referred to *corner23* for short) in the *Geinimi* family. As illustrated in Fig. 1, SFCG contains thousands of function nodes. To identify malware, we merely focus on the sensitive APIs governed by Android permissions. Sensitive APIs constitute only a small portion of the total APIs; however, through them, malware can access sensitive information or perform sensitive tasks.

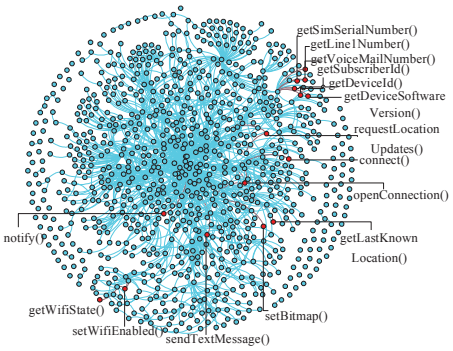


Fig. 1: The SFCG of *corner23*

To obtain the set of sensitive APIs, we use the tool Pscout [36] proposed by Au *et al.* [37]. A total of 680 sensitive APIs are provided by Pscout and can be presented as $SAS = \{s_1, s_2, ..., s_i, s_{i+1}, ..., s_{680}\}$, where s_i is the name of a sensitive API. Fifteen sensitive APIs are used by *corner23*. They are denoted by red nodes with function name labels in Fig. 1. Through these APIs, *corner23* obtains many sensitive resources. For example, it can access the last known location via *getLastKnownLocation()* and can send messages via *sendTextMessage()*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

The contribution of each sensitive API to detecting malicious apps should differ. Thus, a sensitivity coefficient is calculated for each sensitive API in a specific category based on a TF-IDF-like measure to denote its malware detection ability, as described in Section IV-A.

In our two proposed assumptions, for piggybacked apps, the rider and carrier have different invocation patterns of sensitive APIs, which can be used as the basis of our approach. However, analyzing the entire graph is neither effective (the malicious part is buried in the app code) nor efficient (too many nodes and edges to analyze). Mining a representative invocation structure from SFCG can help understand the most suspicious behavior of the given app. For piggybacked apps, the maliciousness and cohesion of sensitive APIs in the invocation structures mined from them would be higher than those in benign apps. Therefore, with the identified sensitive APIs, SFCG can be divided into a set of subgraphs with the sensitive API nodes and their nearby normal nodes. The process of constructing the subgraph set (SGS) is described in Section IV-B. The subgraph with the highest sensitivity coefficient is selected as the indicator of the maliciousness of the app. We call this subgraph the sensitive subgraph.

Definition 2 Sensitive subgraph (SSG). SSG is a subgraph in SGS, and it has the highest sensitivity coefficient among all subgraphs in SGS. It can be obtained with Eqs. (1) and (2).

$$SSG = \underset{SG_j \in SGS}{argmax}\ (scg(SG_j)), \tag{1}$$

$$scg(SG_j) = \sum_{s_i \in SNG(SG_j)} scs(s_i), 1 \leq j \leq m. \tag{2}$$

- $scg(SG_j)$ is the sensitivity coefficient of subgraph SG_j.
- $SNG(SG_j)$ is the set of sensitive APIs contained in SG_j.
- $scs(s_i)$ is the sensitivity coefficient of sensitive API s_i.
- m is the number of subgraphs in SGS.

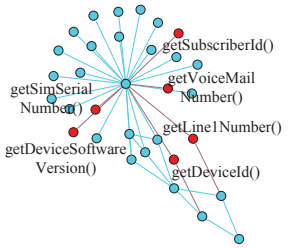


Fig. 2: The SSG of *corner23*

Fig. 2 shows the extracted SSG of *corner23*. The SSG consists of six sensitive APIs and nearby normal nodes. By manually analyzing the code, we find that the SSG extracted from *corner23* is located in the most notorious module of the *Geinimi* family. The module is used to collect users' sensitive information every five minutes, such as the device ID via *getDeviceId()* and the phone number via *getLine1Number()*.

IV. DAPASA

As shown in Fig. 3, DAPASA consists of four steps. First, the apk file is given as the input, whose classes.dex file [38] is converted into .smali files (an interpreted language that syntactically approaches pure source codes) with apktool. By scanning the .smali files, the possible functions and the calling relations between them can be obtained. Thus, SFCG can be constructed in manner in which nodes denote the functions and edges denote the calls. Second, two key steps are performed: measuring the sensitivity coefficient of each sensitive API and mining the SSG in the generated SGS. Lastly, five features of SSG are constructed and fed into machine learning algorithms to detect whether the app is piggybacked or benign.

A. Measurement of the Sensitivity Coefficient

The sensitivity coefficient is calculated to denote the maliciousness of a sensitive API in performing malicious behavior. Given that several sensitive APIs are used in malware and benign apps, the measurement would be biased if only the coefficient of a sensitive API is calculated with its frequency of occurrence in a malicious dataset, such as MIGDroid [39].

We propose a TF-IDF-like measure of the sensitivity coefficient of sensitive APIs that exploits the idea of TF-IDF [40], [41]. To achieve this, 6,154 malicious apps are downloaded from VirusShare [42], and 44,921 benign apps in 26 categories, such as *Game*, *Personalization*, and *Weather*, are collected from Google Play and Anzhi Market. We use six terms of sensitive API s_i to understand its distribution in our malicious and benign datasets.

- $mc(s_i)$: malicious count of s_i. It denotes the number of malware using s_i in the malicious dataset.
- $bc(s_i, c)$: benign count of s_i. It denotes the number of benign apps using s_i in category c.
- $mrt(s_i)$: ratio of $mc(s_i)$ to the total number of malware in the malicious dataset which is represented as p. $mrt(s_i)$ can be obtained with $mrt(s_i) = \frac{mc(s_i)}{p}$, where $p = 6,154$ in our work.
- $brt(s_i, c)$: ratio of $bc(s_i, c)$ to the total number of benign apps in category c which is represented as $q(c)$. $brt(s_i, c)$ can be obtained with $brt(s_i, c) = \frac{1+bc(s_i,c)}{q(c)}$.
- $mrk(s_i)$: rank number of $mrt(s_i)$ among all the sensitive APIs.
- $brk(s_i, c)$: rank number of $brt(s_i, c)$ among all the sensitive APIs in category c.

TABLE I shows several sensitive APIs with high mrts and their corresponding brts and brks in *Game*, *Personalization*, and *Weather* categories, respectively. We obtain three observations from TABLE I.

(1) Several sensitive APIs are used frequently in the malicious and benign datasets. For example, *openConnection()* and *connect()* are used to connect the Internet. Regardless of the category, their brks are very small.

(2) Several sensitive APIs are used more frequently in the malicious dataset than in the benign dataset. An example is *sendTextMessage()*. Its mrk is 2, but its brks in all the three categories exceed 50.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

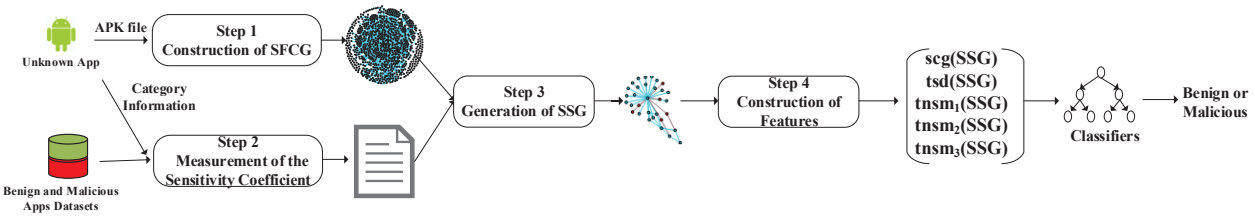


Fig. 3: Overview of DAPASA

TABLE I: SEVERAL SENSITIVE APIs' mrts, mrks, AND THEIR CORRESPONDING brts,brks, scss, AND $rank$s IN *Game*, *Personalization* AND *Weather* CATEGORIES

Sensitive API	Malicious Dataset		Game				Personalization				Weather			
	mrt	mrk	brt	brk	scs	rank	brt	brk	scs	rank	brt	brk	scs	rank
notif()	0.746	1	0.201	23	0.521	2	0.212	4	0.501	9	0.465	14	0.246	7
sendTextMessage()	0.542	2	0.034	50	0.792	1	0.008	65	1.130	1	0.009	80	1.102	1
openConnection()	0.479	3	0.799	1	0.047	46	0.297	1	0.252	21	0.838	1	0.037	43
getDeviceId()	0.474	4	0.456	10	0.161	20	0.085	13	0.508	8	0.414	16	0.181	12
getLine1Number()	0.452	5	0.110	32	0.432	3	0.016	43	0.807	2	0.014	65	0.839	2
connect()	0.449	6	0.747	3	0.057	43	0.216	3	0.299	16	0.738	2	0.059	35
getInputStream()	0.344	9	0.263	18	0.200	18	0.051	23	0.447	11	0.516	11	0.099	24
getSubscriberId()	0.344	10	0.073	39	0.391	4	0.012	50	0.657	3	0.275	31	0.192	11
getConnectionInfo()	0.320	11	0.178	25	0.240	9	0.018	40	0.561	5	0.074	35	0.362	4
getSimSerialNumber()	0.285	14	0.050	44	0.371	5	0.010	58	0.575	4	0.014	64	0.529	3
getActiveNetworkInfo()	0.260	18	0.747	2	0.033	51	0.279	2	0.144	32	0.738	3	0.034	44
getLastKnownLocation()	0.217	21	0.418	13	0.082	36	0.079	14	0.239	23	0.595	8	0.049	38
requestLocationUpdates()	0.181	22	0.267	17	0.203	17	0.057	20	0.232	24	0.599	7	0.063	32
getCellLocation()	0.152	26	0.023	54	0.249	8	0.004	79	0.364	12	0.039	43	0.214	8

(3) The brts and brks differ in the different categories. For example, in categories *Game* and *Weather*, nearly all sensitive APIs have higher brts than those in the category *Personalization*, which have lower than 0.3.

With these three observations, we consider the following questions to better understand our measurement of the sensitivity coefficient.

Q 1 ***If the mrt of a sensitive API is high, will its sensitivity coefficient also be high?***

As illustrated in TABLE I, the mrts of *openConnection()* and *getSimSerialNumber()* (used to obtain the user's SIM number) are 0.479 and 0.285, respectively. Does this mean *openConnection()* has a higher sensitivity coefficient than *getSimSerialNumber()*?

The answer is no. As noted in the first two observations, *openConnection()* is widely used in both malicious and benign apps because nowadays, most apps need to connect to the Internet. Meanwhile, *getSimSerialNumber()* occurs much more frequently in malicious apps than in benign apps because benign apps rarely need to have the SIM number. Intuitively, *getSimSerialNumber()* should have a higher sensitivity coefficient than *openConnection()*.

Q 2 ***Does an app that obtains location information by using getLastKnownLocation() appear suspicious?***

The answer is also no. As noted in the last observation, the brts in the three categories are different. For apps in the *Personalization* category, the API's brt is only 0.079 and would reveal the location information of users. In the *Weather* category, the API's brt is 0.595, and the API is generally used to obtain weather information in the location of users. According to this discussion, the category information can be exploited in our measurement of sensitivity coefficients. For the same sensitive API, its sensitivity coefficients in different categories would be different.

In text mining literature, TF-IDF is a numerical statistic intended to reflect how discriminating a term is to a document in a corpus. By utilizing the idea of TF-IDF for reference, we make the scs of a sensitive API be in positive correlation with its mrt and in negative correlation with its brt. For sensitive API s_i of an app that belongs to a specific category c, its sensitivity coefficient $scs(s_i)$ is calculated with Eq. (3).

$$scs(s_i) = mrt(s_i) \times \log \frac{1}{brt(s_i, c)}. \quad (3)$$

For example, the *Game* category has 3,505 apps, in which 2,801 apps use *openConnection()* and 174 apps use *getSimSerialNumber()*. Their scss are 0.047 and 0.371, respectively. Apparently, *getSimSerialNumber()* is more sensitive than *openConnection()*.

TABLE I also shows the scss and $rank$s of the sensitive APIs. *sendTextMessage()* has the highest scs in all the three categories, given that it is frequently used by malicious apps and rarely used by benign apps. This condition reflects the common attack of stealthily sending SMS messages to premium numbers, thus allowing the owner of these numbers to earn money from the victims. Combined with sending SMS messages, the sensitive APIs utilized to obtain the user's privacy information, such as phone number (*getLine1Number()*) and SIM number (*getSimSerialNumber()*), would also have high coefficients. Unlike the previous ones, sensitive APIs

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

used frequently both in malicious and benign apps, such as *openConnection()*, are assigned with low coefficients.

The results show that the sensitivity coefficients calculated by the TF-IDF-like measure can reflect the maliciousness of sensitive APIs in different categories.

However, there are some apps that have no category information, especially the malware samples downloaded from VirusShare. We calculate the sensitivity coefficients of sensitive APIs for such apps as:

$$scs(s_i) = mrt(s_i) \times \log \frac{1}{brt(s_i)}. \quad (4)$$

$brt(s_i)$ denotes the percent of apps in all benign apps using the sensitive API s_i and it is obtained with Eq. (5), in which C denotes the set of all the benign categories.

$$brt(s_i) = \frac{1 + \sum_{c \in C} bc(s_i, c)}{\sum_{c \in C} q(c)} \quad (5)$$

B. Generation of SSG

Based on our proposed assumptions, SFCG is divided into a set of subgraphs, and the subgraph that has the highest sensitivity coefficient is selected as SSG, which can profile the suspicious behavior of the given app. SSG can be generated through the following steps.

1) Generation of SGS : Algorithm I highlights the step of generating the subgraph set with the input of the SFCG of a given app and its invoked sensitive API node set (SS). For each sensitive API node, a subgraph is constructed with its neighbor nodes in the SFCG. The function $dis(v_k, v_i)$ returns the shortest path length from node v_k to node v_i. When calculating the distance between two nodes, the SFCG is regarded as an undirected graph. In our work, the average shortest path length of the SFCGs is generally from 3 to 5. When constructing subgraphs, the distances of normal nodes to the sensitive API node are less than or equal to 2.

The function $RemoveLibNodes(V_i)$ in algorithm I is utilized to remove the nodes invoked by third-party libraries for V_i via a library list. In our approach, the potential suspicious libraries might cause false positives since such libraries contain similar invocation patterns of sensitive APIs as the malicious payloads do. Therefore, we leverage the result provided by the tool LibD [43] proposed by Li *et al.* [44], which identified 60,729 different third-party libraries with a manually validated accuracy rate. More precisely, we first add the package names of the identified libraries into a list. We then remove the method nodes that are invoked by such packages according to the list from the function call graph. The test result shows that about 81.8% apps in our dataset contain the third-party libraries such as *com/google/ads*, *com/facebook* and *com/umeng*. After this procedure, we can effectively filter the potential suspicious libraries.

2) Selection of SSG: Algorithm II highlights the step of selecting SSG from the SGS generated by algorithm I. In SGS, two subgraphs that contain the same sensitive API nodes may exist. Algorithm II merges the subgraphs with the condition $SNG(SG_i) \cap SNG(SG_j) \neq \emptyset$ to ensure that one sensitive API node can only occur in one subgraph. Afterward, the sensitivity coefficient for each $SG_j \in SGS$ is calculated with Eq. (2), and the subgraph with highest coefficient among all the subgraphs in SGS is selected as the SSG with Eq. (1). If no sensitive API call exists in a given app, then it does not have an SSG.

Algorithm I Generate SGS

Input: $SFCG = \{V, E\}; SS$
Output: SGS

```
SGS ← ∅;
for each v_i ∈ SS do
    V_i ← ∅;
    for each v_k ∈ V do
        if dis(v_k, v_i) <= 2 then
            V_i = V_i ∪ {v_k};
        end if
    end for
    RemoveLibNodes(V_i);
    E_i = V_i × V_i ∩ E;
    SG_i ← (V_i, E_i);
    SGS = SGS ∪ {SG_i};
end for
return SGS
```

Algorithm II Select SSG from SGS

Input: SGS
Output: SSG

```
while ∃SG_i, SG_j ∈ SGS, i ≠ j and SNG(SG_i) ∩ SNG(SG_j) ≠ ∅ do
    V_j = V_i ∪ V_j, E_j = E_i ∪ E_j
    SGS = SGS \ {SG_i}
end while
SSG = argmax_{SG_j ∈ SGS} (scg(SG_j))
return SSG
```

C. Construction of Features

By employing SSG, we construct a set of features from SSG based on our two proposed assumptions. The features fall into three fields to distinguish piggybacked apps from benign apps in different aspects. We randomly select 500 piggybacked apps and 500 benign apps, respectively, to determine if our features are able to distinguish them.

1) Sensitivity Coefficient of SSG—$scg(SSG)$: $scg(SSG)$ is defined to denote the maliciousness of SSG. As mentioned in assumption I, to perform its malicious task, the rider would make many sensitive API calls; thus, the maliciousness of SSG of piggybacked app is higher than that of benign app.

As illustrated in Fig. 4, the median of the coefficients of piggybacked apps is 1.341, which is higher than that of benign apps (0.444) because they have fewer invocations of sensitive APIs. This result proves that our assumption I is tenable. Obviously, $scg(SSG)$ can effectively distinguish piggybacked apps from benign ones.

2) Total Sensitive Distance of SSG—$tsd(SSG)$: As mentioned in assumption II, the cohesion of sensitive APIs in the rider is generally higher than that in the carrier. We use

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

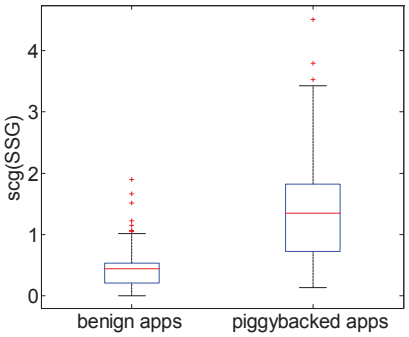


Fig. 4: $scg(SSG)$ for benign and piggybacked apps

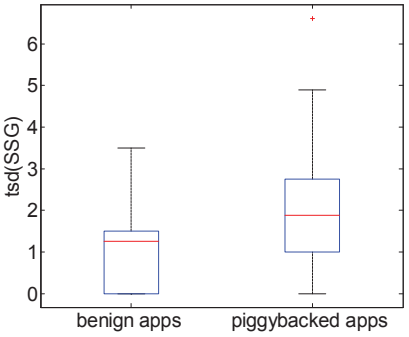


Fig. 5: $tsd(SSG)$ for benign and piggybacked apps

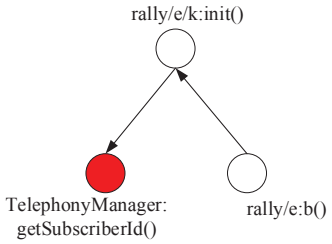


Fig. 6: An instance of sensitive motif-2 in SSG

$tsd(SSG)$ to denote the cohesion of sensitive API nodes in SSG, which is measured by the calling distances between sensitive API nodes.

$tsd(SSG)$ can be obtained with Eqs. (6) and (7), in which $sd(s_i)$ denotes the average distance of sensitive API node s_i to the other sensitive API nodes in SSG.

$$tsd(SSG) = \sum_{s_i \in SNG(SSG)} sd(s_i). \quad (6)$$

$$sd(s_i) = \frac{1}{|SNG(SSG)|-1} * \sum_{\substack{s_j \in SNG(SSG) \\ j \neq i}} \frac{1}{dis(s_i,s_j)}. \quad (7)$$

As illustrated in Fig. 5, the median of $tsd(SSG)$ of piggybacked apps is 1.875, which is even higher than the upper quartile of benign apps. This result indicates that assumption II is tenable. Thus, the feature $tsd(SSG)$ is effective to distinguish piggybacked apps from benign ones.

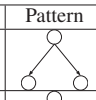
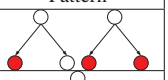
3) Total Number of Sensitive Motif Instances in SSG—$tnsm(SSG)$: We have attempted to obtain a more detailed view of the invocation patterns between sensitive API nodes and normal nodes.

An invocation pattern reflects one malicious behavior of an app, which can be depicted by a motif. Network motifs are defined in terms of connectivity-patterns that appear much more often than expected from pure chance [45]–[47]. Specifically, they occur at a higher frequency than what is expected from an ensemble of randomized graphs with an identical degree structure. Given that no mutual edges exist in SSG, four three-node motifs are present. The four three-node motifs and their average Z-score values in our samples are shown in TABLE II with the help of gtrieScanner [48]. The higher the Z-score is, the more significant the three-node pattern is as a motif. The Z-score of motif-4 is less than 0, which means that it rarely occurs in SSG. Thus, it is ignored in our computation.

Sensitive motifs are defined in this work as significant motifs that contain at least one sensitive API node. They are shown in TABLE II. For example, the instance of sensitive motif-2 in Fig. 6 denotes the malicious behavior of obtaining the unique subscriber ID number by using an object *rally/e* and invoking the *getSubscriberId()* API.

Under assumptions I and II, because of the larger number and higher cohesion of sensitive APIs in the rider than in the carrier, more instances of sensitive motif-1 occur in SSG. In addition, in the rider, the sensitive APIs are invoked by many user-defined threatening functions, which cause many instances of sensitive motif-2 and sensitive motif-3. We use $tnsm_k(SSG), k = 1, 2, 3$, to denote the total number of sensitive motif-k instances in SSG. Fig. 7 illustrates $tnsm_k(SSG)$ for our benign and piggybacked apps, which demonstrates that for all the three types of sensitive motifs, the corresponding $tnsm_k(SSG)$ for piggybacked apps are higher than those for benign apps.

TABLE II: THREE-NODE MOTIFS AND THEIR CORRESPONDING SENSITIVE MOTIFS

Three-node Motifs			Sensitive Motifs	
Index	Pattern	Z-score	Index	Pattern
motif-1		1.349	sensitive motif-1	
motif-2		1.356	sensitive motif-2	
motif-3		1.308	sensitive motif-3	
motif-4		-0.499		

The features constructed from the SSGs of piggybacked apps differ significantly from those of benign apps. DAPASA embeds the above five features into a feature space to automatically classify novel apps as piggybacked apps or not. The feature space is represented as follows:

$$\begin{pmatrix} scg(SSG) \\ tsd(SSG) \\ tnsm_1(SSG) \\ tnsm_2(SSG) \\ tnsm_3(SSG) \end{pmatrix}. \quad (8)$$

V. Evaluation

To evaluate the effectiveness of our approach, we first introduce the dataset and the metrics (see Section V-A for details). We then evaluate our approach based on the dataset and compare the result with that of three baseline approaches (see Section V-B for details). Afterward, we evaluate the runtime overhead of our approach and compare it with that of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

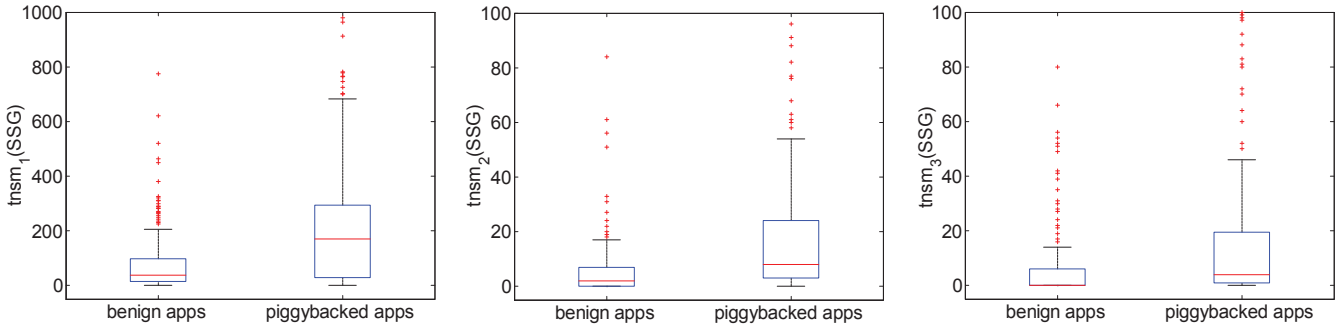


Fig. 7: $tnsm(SSG)$ for benign and piggybacked apps

the baseline approaches (see Section V-C for details). Finally, we analyze the effectiveness of our features and how they complement the permission-based approaches and API-based approahces (see Section V-D for details).

A. Dataset and Metrics

Our approach is evaluated on a large real-world dataset that consists of Android benign apps and piggybacked apps.

The set of piggybacked apps contains 2,551 apps in 15 families. All the apps are piggybacked apps according to [4]. A total of 1,062 of the apps are downloaded from the Android Malware Genome Project [49], which is widely used as a benchmark dataset for malware detection. We collect 1,489 more piggybacked apps that belong to the piggybacked families [4] from VirusShare based on our malware familial classification approach that classifies each unlabeled malware into its corresponding family with a 96% classification accuracy [50]. An overview of the piggybacked apps in our dataset is given in TABLE III.

TABLE III: DESCRIPTIONS OF THE PIGGYBACKED APPS

Family	#apps	Family	#apps
Adrd	55	FakeInstaller	769
AnserverBot	187	Geinimi	94
BaseBridge	122	GingerMarster	350
BeanBot	8	GoldDream	46
Bgserv	9	HippoSMS	13
DroidDream	49	Jifake	41
DroidDreamLight	46	Pjapps	66
DroidKungFu	696	**Total**	**2,551**

The set of benign apps consists of two parts; one is collected from Google Play and contains 12,001 apps in 16 categories, and the other one is collected from Anzhi Market and contains 32,920 apps in 10 categories. TABLE IV shows the descriptions of apps from Google Play and Anzhi Market. All the apps have been checked by VirusTotal [51] to ensure that each of them is benign. Over 50 anti-virus softwares programs, such as AVG [52], ESET-NOD32 [53] and Norton [54], are available in VirusTotal; these software programs are based on a signature database. They are useful for known malware but less effective for unknown ones.

The metrics used to measure our detection results are shown in TABLE V. The goal of any malware detection research is to achieve a high value for TPR and a low value for FPR. We conduct the experiments in over 4,000 lines of Java code on a quad-core 3.20 GHz PC operating on Ubuntu 14.04 (64 bit) with 16 GB RAM and 1 TB hard disk.

TABLE IV: DESCRIPTIONS OF THE BENIGN APPS

Google Play Apps			Anzhi Market Apps		
Id	Category	#apps	Id	Category	#apps
GA	Business	491	AA	Communication	1,122
GB	Comics	497	AB	Finance	2,177
GC	Communication	622	AC	Music&Audio	1,307
GD	Education	577	AD	News Reading	2,400
GE	Entertainment	798	AE	Office Work	4,970
GF	Finance	492	AF	Shopping	4,837
GG	Game	3,505	AG	Social	3,265
GH	Lifestyle	789	AH	System Tools	3,150
GI	Medical	374	AI	Themes Desktop	7,962
GJ	Personalization	732	AJ	Weather&Travel	1,730
GK	Photography	491		**Total**	**32,920**
GL	Productivity	569			
GM	Shopping	377			
GN	Social	630			
GO	Tools	625			
GP	Weather	432			
	Total	**12,001**			

TABLE V: DESCRIPTIONS OF THE USED METRICS

Term	Abbr	Definition
True Positive	TP	#malicious apps classified as malicious apps
True Negative	TN	#benign apps classified as benign apps
False Negative	FN	#malicious apps classified as benign apps
False Positive	FP	#benign apps classified as malicious apps
True Positive Rate	TPR	TP/(TP+FN)
False Positive Rate	FPR	FP/(FP+TN)
Precision	p	TP/(TP+FP)
Recall	r	TP/(TP+FN)
F-measure	F_1	2rp/(r+p)
ROC Area	AUC	Area under ROC curve

B. Piggybacked App Detection

1) Detection Performances with Four Classifiers: Four different classifiers are employed to evaluate our approach. These classifiers are Random Forest [55], Decision Tree (C4.5) [56], k-NN(k=1) [57] and PART [58]. All the 49,921 benign apps and 2,551 piggybacked apps are mixed together. After the extraction and analysis of the SSGs with our approach, each

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

app is first represented as a feature vector with Eq. (8). Then the classification labels of the known piggybacked apps in training dataset are attached with 1 while the labels of the known benign apps are attached with -1 so that the classifiers can understand the discrepancy between piggybacked apps and benign apps. Once the feature vectors with classification labels for the training samples are generated, four classifiers can be trained with the four machine learning algorithms. After that, the feature vector of a new sample without classification label is fed into the classifiers to detect whether it is piggybacked or benign. Our dataset is evaluated via tenfold cross validation.

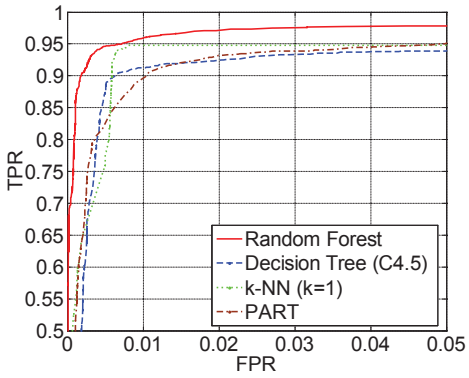


Fig. 8: Detection performances with four different classifiers

The detection performances are shown in Fig. 8. The Receiver Operating Characteristic (ROC) curves indicate that all four classifiers can achieve a high value for TPR and a low value for FPR. In particular, Random Forest performs best among four classifiers. With Random Forest, the detection performance yields a TPR of 0.950 at an FPR of 0.007, and the AUC is 0.99.

Two main reasons explain the best performance of Random Forest in the current study's dataset. First, Random Forest is an ensemble classifier that uses out-of-bag errors as an estimate of the generalization error to improve its performance, whereas the other three classifiers are base classifiers. Second, as introduced in the work of Breiman [55], Random Forest does not result in overfitting as more trees are added but produces a limited value of the generalization error. Therefore, in this work, Random Forest is selected as the classifier in subsequent experiments.

2) Comparison with Three Baseline Approaches: In this section, DAPASA is compared with three baseline approaches proposed by Wang *et al.* [16], Aafer *et al.* [33], and Gascon *et al.* [26]. The descriptions of the three baseline approaches are shown below.

- Wang *et al*. [16] proposed an approach for malware detection based on *requested permissions*, which are security-aware features that restrict the access of apps to the core facilities of devices.
- Aafer *et al*. [33] proposed an approach for malware detection based on *APIs* that have more fine-grained features than permissions because each permission governs several APIs. Furthermore, API level information conveys more substantial semantics about the app than permissions [33].
- Gascon *et al*. [26] proposed an approach for malware detection based on *embedded call graphs*, which model the structural composition of a code and reflect the logic semantics of the app. The call graph is more robust against certain obfuscation strategies than the requested permissions and APIs.

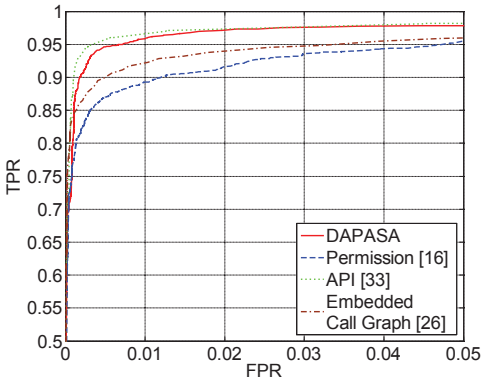


Fig. 9: Detection performances for DAPASA and three baseline approaches

The detection performances of our approach and the three baseline approaches in our dataset are illustrated in Fig. 9. The AUC values of our approach and API-based approach [33] are both 0.99, which indicates that our approach has a similar detection performance with API-based approach. Moreover, our approach outperforms the other two baseline approaches [16], [26], of which the AUC values are 0.983 and 0.986, respectively. In particular, our approach contains only five numeric features; the three approaches use 88 permission-based features, 680 API-based features, and 32,768 graph-based features, respectively.

C. Evaluation of Run-time Overhead

1) Run-time Overhead of DAPASA: Our approach consists of three main procedures when analyzing a new app.

(i) **De-compilation.** The app file is disassembled to generate the Dalvik code, and SFCG is constructed.
(ii) **Graph analysis.** The SFCG is divided into a set of subgraphs, and the SSG with the highest sensitivity coefficient is selected.
(iii) **Feature construction.** Five numeric features are constructed from the generated SSG.

The run-time overheads of the three main procedures and their total run-time overhead are illustrated in Fig. 10, in which the *x*-axis shows the sample size (number of nodes) per app in our dataset and the *y*-axis shows the run-time overhead of the corresponding procedure.

Four observations are obtained from Fig. 10.

(1) The run-time overhead of de-compilation is not related to the sample size. This result is consistent with the truth that the complexity of de-compilation has a positive correlation with the logic of the source code for a given app rather than the sample size [59].

(2) The run-time overhead of graph analysis roughly scales linearly with the sample size. As introduced in algorithm I, the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

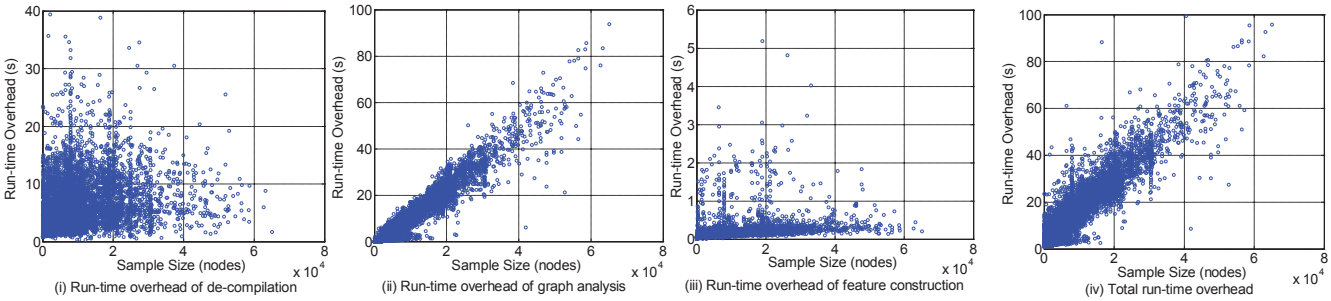


Fig. 10: Run-time overhead of DAPASA

time complexity is $O(m \times n)$, where m denotes the number of invoked sensitive API nodes and n denotes the size of the call graph.

(3) The run-time overhead of feature construction is not related to the sample size. In our approach, SSG is generated to represent the entire call graph. Therefore, the run-time overhead of feature construction scales with the size of SSG rather than the size of the call graph.

(4) The total run-time overhead of analyzing a given app has positive relation with sample size. It is mainly affected by the procedure of de-compilation with a relatively small sample size. With the increase in sample size, the total run-time overhead is mainly affected by the graph analysis procedure. On the average, less than 16s is consumed to complete the analysis for most apps in our dataset.

2) Comparison of Run-time Overhead: The comparison of the run-time overheads of our approach and the three baseline approaches is illustrated in Fig. 11. DAPASA consumes 1.8s and 4.6s less time than the approach of Gascon *et al.* [26] in graph analysis and feature construction, respectively. The smaller run-time overhead is due to the following reasons.

First, for the graph analysis procedure, in the approach of Gascon *et al.* [26], a hash-value is calculated for each node in the graph. Analyzing all the nodes consumes more time than our approach does because our approach only focuses on the analysis of sensitive API nodes.

Second, for the feature construction procedure, in the approach of Gascon *et al.* [26], a feature map is inspired by graph kernels, which allows for embedding call graphs in a vector space. However, our approach generates SSG to represent the entire call graph. Hence, computational complexity is reduced effectively.

The approaches of Wang *et al.* [16] and Aafer *et al.* [33] do not have the graph analysis procedure. Therefore, they are faster than DAPASA and the approach of Gascon *et al.* [26], which are based on the analysis of the call graph. Permission-based and API-based approaches usually produce only a small run-time overhead, and they are efficient and scalable. However, the features of permissions and APIs are coarse-grained. For example, malicious apps may request the exact same permissions that are requested by benign apps. By contrast, our features are more fine-grained and thus provide better explanation of the results, as discussed in Section VI-B.

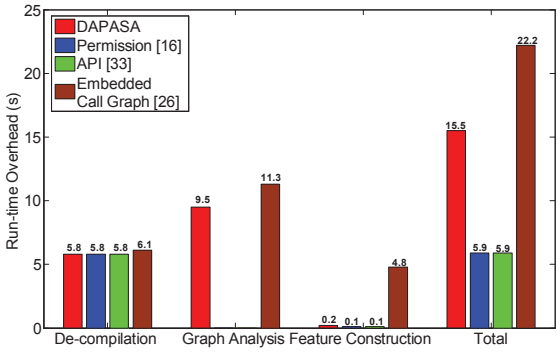


Fig. 11: Comparison results of run-time overhead

D. Analysis of Features

1) Effectiveness of Each Feature: In this work, we propose three different types of features, namely, scg, tsd, and $tnsm$ (consisting of $tnsm_1$, $tnsm_2$ and $tnsm_3$) to distinguish the SSGs existed in piggybacked apps from those existed in benign apps. As mentioned in Section IV-C, each of them has a fairly good ability to detect piggybacked apps in different aspects, such as maliciousness and cohesion of sensitive APIs. In this section, different combinations of features are evaluated in the same dataset to determine whether each feature is significant for the detection performance. Only scg is initially used as our feature. Afterward, the other two types of features are added to our feature space successively.

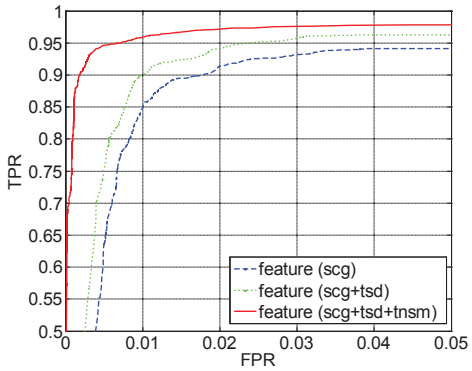


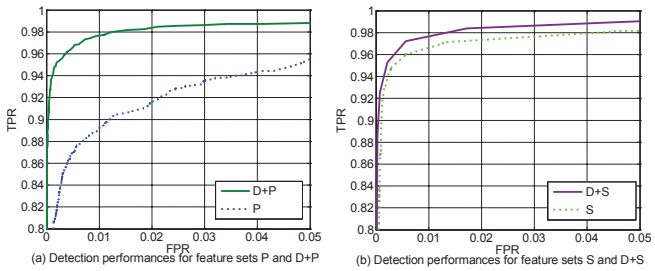
Fig. 12: Detection performances with different feature combination

As illustrated in Fig. 12, the ROC curves with different

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

feature combinations show that every additional feature effectively improves the detection performance. The TPR reaches nearly 0.85 with an 0.01 FPR using only scg, and it is improved by 0.05 and 0.061 by adding tsd and $tnsm$. The improvements of TPRs demonstrate that each proposed feature has significant contributions for piggybacked app detection.

2) Complementation of Existing Approaches: Five features are constructed from a new perspective of the invocation structure. We combine five features with the permission-based features proposed by Wang *et al*. [16] and API-based features proposed by Asfer *et al.* [33], respectively. The detection performances of the four different feature sets are illustrated in Fig. 13, in which *P* denotes the 88 permissions, *S* denotes the 680 APIs, *D+P* denotes the combination of our five features with permissions, and *D+S* denotes the combination of our five features with APIs.



(a) Detection performances for feature sets P and D+P (b) Detection performances for feature sets S and D+S

Fig. 13: Detection performances for different feature sets

As illustrated in Fig. 13, after the combination of our five features, the detection performances measured with ROC curves are improved. When the FPR is set to 0.01, the TPRs with feature *D+P* and with feature *D+S* are improved by 0.085 and 0.008 compared with those with only feature *P* and feature *S*, respectively. Therefore, our approach can complement the permission-based approaches and API-based approaches from a new perspective of the invocation structure.

Moreover, the contribution degrees of our five features are evaluated with three different types of metrics, namely, chi-square statistic [60], OneR classifier [61], and information gain [62], for the two combined feature sets (*D+P* and *D+S*) containing 93 and 685 features, respectively. The result in TABLE VI shows that the five features have better contributions to classify piggybacked apps than most permission-based and API-based features especially scg and $tnsm_1$.

TABLE VI: FEATURE RANKING OF OUR FEATURES IN THE FEATURE Feature *D+P* AND FEATURE *D+S*

Feature	Chi-squared Statistic		OneR Classifier		Information Gain	
	D+P	*D+S*	*D+P*	*D+S*	*D+P*	*D+S*
scg	1	1	1	1	1	1
tsd	3	3	3	3	4	3
$tnsm_1$	2	2	2	2	2	2
$tnsm_2$	9	8	13	15	8	9
$tnsm_3$	8	6	4	5	7	7

VI. DISCUSSIONS

In this section, we first inspect the reasons of the generation of false positive instances. Then we introduce the explanations of the detection results for DAPASA. After that, the ability of DAPASA to fight against obfuscation attacks is discussed. Finally, we present some limitations of our approach.

A. Discussions on TPR and FPR

The experiments show that our approach achieves good performance with a TPR of 95% and an FPR of 0.7%. Manual analysis of the SSGs of our piggybacked apps shows that DAPASA achieves a 100% detection rate in several families, such as *Geinimi*. The invocation patterns of sensitive APIs in the generated SSGs for all the *Geinimi* samples are exactly the same as that of the example introduced in Section III. However, the TPRs are lower than 92% in several families, such as *DroidKungFu*, which is considered one of the most sophisticated Android malware. *DroidKungFu* is piggybacked and distributed in the forms of legitimate apps. Several samples implement their malicious functionalities in native code (instead of the previously Davilk code based on Java). In this work, the native code is ignored, thus resulting in the lower accuracy for such families.

Although the TPR is impressive, the FPR is 0.7% which means that there are still more than 300 benign apps are incorrectly classified as piggybacked apps. Two main reasons explain the incorrectly classified samples. First, with the help of LibD [44] we are able to remove most nodes invoked by third-party libraries with a string matching algorithm. However, covering all third-party libraries is still a challenge. Second, several extreme cases which are repackaged with only one sensitive API (*sendTextMessage*) exist. When these extreme cases are placed in the training dataset, the benign apps using the same sensitive API as the extreme cases do might be classified as piggybacked. For example, in the *Game* category, sending a message to a premium number to raise money is a legitimate payment method for unlocking game features, and the apps that use this method would be incorrectly classified.

B. Explanations of the Detection Results

Explanations of the detection results are also important for malware detection. For permission-based and API-based approaches, even if they usually produce only a small runtime overhead and are efficient and scalable, they do not provide reasonable explanations for their decisions and are thus unclear to the practitioner. However, DAPASA is based on the analysis of the call graph that contains the necessary structure information to depict app behaviors. DAPASA constructs an SSG to profile the most suspicious behavior of the given app, thus providing better explanations of the decisions than permission-based and API-based approaches. For example, in our experiments, all the SSGs constructed from the 94 members of the *Geinimi* family are nearly the same. More concretely, we introduce the meanings of our constructed features for SSG in Fig. 2.

$scg = 1.404$. Six sensitive APIs are in the subgraph; three of them (*getLine1Number(), getSubscriberId()*, and *getSimSerialNumber()*) have high scss and low $rank$s, as shown in TABLE I. Its scg is much higher than the median in benign apps, which is only 0.444 (illustrated in Fig. 4). The descriptions

of the six invoked sensitive APIs indicate that the module in which SSG is located is probably used to collect user's sensitive information.

$tsd = 3$. All the calling distances between sensitive API nodes are 2; this means that they are continually invoked in the same method to collect user's sensitive information. This conclusion is demonstrated in the corresponding source code of the app.

Each sensitive motif has its own meaning. For example, the instance of sensitive motif-2 in Fig. 6 denotes the malicious behavior of obtaining the unique subscriber ID number by using an object *rally/e* and invoking the *getSubscriberId()* API. In summary, the instances of sensitive motifs illustrate the detailed invocations of sensitive APIs of an app.

C. Resilience to Obfuscation Attacks

In order to evaluate whether our approach could be robust against obfuscation attacks mentioned in [63], we conduct two widely used tools, Proguard [64] and SandMarks [65] introduced in [63], to obfuscate the APK samples. Proguard is able to rename the classes, fields, and methods using short meaningless names. SandMarks is a very comprehensive tool, which implements 39 obfuscation algorithms, such as constant pool reorder, reorder parameters and method merger. We then calculate the similarities of generated SSGs between the original apps and obfuscated apps. The similarities are still 1 which demonstrates that our approach is robust against the typical obfuscation attacks (e.g., renaming functions). The main reason is that our approach is based on the analysis of sensitive subgraph, which does not consider the names of methods and parameters. However, the advanced obfuscation attacks that change the invocation relations among functions have side effects for our approach.

D. Limitations of DAPASA

Similar to any empirical approach, our approach is subject to several limitations, which are listed below.

Encryption and reflection. By analyzing the call graph of the app, our approach is resilient to typical local obfuscation techniques [63], [66], such as renaming of the user-defined functions and packages, instruction reordering, and branch inversion. However, it is vulnerable to certain obfuscation techniques, such as encryption [67], [68] and refection [15]. Once the malware code is encrypted, it is difficult to obtain the source code of the app with decompile tools and construct the call graph. In addition, the reflection techniques can simply hide away the edges in the call graph, such as the invoking method with the function *getMethod(String name)* where the argument *name* denotes the name of the callee method.

Sensitive APIs. Our detection of sensitive APIs relies on the mapping by PScout [37], which now, five years later, may be partially outdated. Incorrect or missing entries in the mapping would make DAPASA miss or misclassify relevant app behaviors.

VII. CONCLUSION AND FUTURE WORK

In this work, we proposed DAPASA that focuses on piggybacked app detection through sensitive subgraph analysis. First, two assumptions were proposed to better profile the differences between the rider and carrier in piggybacked apps with respect to the invocation patterns of sensitive APIs. Second, an SSG was generated for each app to profile its most suspicious behavior. Third, five features were constructed from the SSG and fed into machine learning approaches to detect piggybacked apps.

Extensive evaluation results demonstrate that our approach achieves an impressive detection performance with only five numeric features which bring three advantages. First, our approach outperforms the state-of-the-art approaches with less features. Second, our approach provides better explanations of detection results than permission-based approaches and API-based approaches. Third, our approach even complements permission-based approaches and API-based approaches with the combination of our features from a new perspective of the invocation structure.

The work presented in this paper can be improved by building a more detailed behavior model than SFCG. Additional information, such as components of the app and type of invocations, would be required to help improve the detection accuracy of Android piggybacked apps.

VIII. ACKNOWLEDGMENT

This work was supported by National Key Research and Development Program of China (2016YFB1000903), National Natural Science Foundation of China (91418205, 61472318, 61532015, 61532004, 61672419, 61632015), Fok Ying-Tong Education Foundation (151067), Ministry of Education Innovation Research Team (IRT13035), the Fundamental Research Funds for the Central Universities, and Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892).

REFERENCES

[1] "Google play," https://play.google.com/store, 2016.
[2] "Anzhi market," http://www.anzhi.com, 2016.
[3] "Security report of qihoo," http://zt.360.cn/report/, 2016.
[4] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. SP*, 2012.
[5] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proc. CODASPY*, 2013.
[6] G. Suarez-Tangil, J. E. Tapiador, F. Lombardi, and R. Di Pietro, "Thwarting obfuscated malware via differential fault analysis," *IEEE Computer*, vol. 47, no. 6, pp. 24–31, 2014.
[7] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proc. ACSAC*, 2014.
[8] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. WiSec*, 2014.
[9] Q. Guan, H. Huang, W. Luo, and S. Zhu, "Semantics-based repackaging detection for mobile apps," in *Proc. ESoSS*, 2016.
[10] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proc. ISSTA*, 2015.
[11] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *Proc. NDSS*, 2012.

[12] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proc. MobiSys*, 2012.

[13] M. Zheng, M. Sun, and J. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware," in *Proc. TrustCom*, 2013.

[14] 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," *Elsevier Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[15] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.

[16] 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 Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, 2014.

[17] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proc. NDSS*, 2014.

[18] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proc. NDSS*, 2014.

[19] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. ICSE*, 2014.

[20] M. Zhao, F. Ge, T. Zhang, and Z. Yuan, "Antimaldroid: an efficient svm-based malware detection framework for android," in *Proc. ICICA*, 2011.

[21] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proc. CCS*, 2014.

[22] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Elsevier Computers and Security*, vol. 39, pp. 340–350, 2013.

[23] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proc. CIS*, 2011.

[24] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2017.

[25] Z. Tian, T. Liu, Q. Zheng, M. Fan, E. Zhuang, and Z. Yang, "Exploiting thread-related system calls for plagiarism detection of multithreaded programs," *Elsevier Journal of Systems and Software*, vol. 119, pp. 136–148, 2016.

[26] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proc. AISec*, 2013.

[27] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *Proc. ESORICS*, 2014.

[28] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting code reuse in android applications using component-based control flow graph," in *Proc. SEC*, 2014.

[29] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proc. PPREW*, 2014.

[30] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proc. ICSE*, 2014.

[31] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *Proc. Security*, 2015.

[32] V. Moonsamy, J. Rong, and S. Liu, "Mining permission patterns for contrasting clean and malicious android applications," *Elsevier Future Generation Computer Systems*, vol. 36, pp. 122–132, 2014.

[33] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Proc. SecureComm*, 2013.

[34] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *Elsevier Journal of Systems and Software*, vol. 108, pp. 193–210, 2015.

[35] "apktool: A tool for reverse engineering android apk files," https://ibotpeaches.github.io/Apktool/, 2016.

[36] "Pscout," http://pscout.csl.toronto.edu/, 2012.

[37] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proc. CCS*, 2012.

[38] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: toward extracting hidden code from packed android applications," in *Proc. ESORICS*, 2015.

[39] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han, "Migdroid: Detecting app-repackaging android malware via method invocation graph," in *Proc. ICCCN*, 2014.

[40] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, "Interpreting tf-idf term weights as making relevance decisions," *ACM Transactions on Information Systems*, vol. 26, no. 3, p. 13, 2008.

[41] A. Aizawa, "An information-theoretic perspective of tf–idf measures," *Elsevier Information Processing and Management*, vol. 39, no. 1, pp. 45–65, 2003.

[42] "Virusshare," http://virusshare.com/, 2016.

[43] "Libd," https://github.com/IIE-LibD/libd, 2017.

[44] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *Proc. ICSE*, 2017.

[45] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[46] P. Ribeiro and F. Silva, "Querying subgraph sets with g-tries," in *Proc. DBSocial*, 2012.

[47] S. Wernicke and F. Rasche, "Fanmod: A tool for fast network motif detection," *Oxford Univ Press Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, 2006.

[48] "gtriescanner: Quick discovery of network motifs," http://www.dcc.fc.up.pt/gtries/, 2016.

[49] "Android malware genome project," http://www.malgenomeproject.org/, 2012.

[50] M. Fan, J. Liu, X. Luo, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu, "Frequent subgraph based familial classification of android malware," in *Proc. ISSRE*, 2016.

[51] "Virustotal," https://www.virustotal.com/en/, 2016.

[52] "Avg," http://free.avg.com/us-en/homepage, 2016.

[53] "Eset-nod32," https://www.eset.com/us/, 2016.

[54] "Norton," http://cn.norton.com/, 2016.

[55] L. Breiman, "Random forests," *Springer Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[56] S. L. Salzberg, "C4.5: Programs for machine learning," *Springer Machine Learning*, vol. 16, no. 3, 1994.

[57] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Springer Machine Learning*, vol. 6, no. 1, 1991.

[58] E. Frank and I. H. Witten, "Generating accurate rule sets without global optimization," in *Proc. ICML*, 1998.

[59] A. Johnstone, E. Scott, and T. Womack, "What assembly language programmers get up to: control flow challenges in reverse compilation," in *Proc. CSMR*, 2000.

[60] R. L. Plackett, "Karl pearson and the chi-squared test," *JSTOR International Statistical Review*, vol. 51, no. 1, pp. 59–72, 1983.

[61] R. C. Holte, "Very simple classification rules perform well on most commonly used datasets," *Springer Machine learning*, vol. 11, no. 1, pp. 63–90, 1993.

[62] L. E. Raileanu and K. Stoffel, "Theoretical comparison between the gini index and information gain criteria," *Springer Annals of Mathematics and Artificial Intelligence*, vol. 41, no. 1, pp. 77–93, 2004.

[63] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Proc. TRUST*, 2013.

[64] "Proguard," https://www.guardsquare.com/en/proguard, 2016.

[65] "Sandmarks: A tool for the study of software protection algorithms," http://sandmark.cs.arizona.edu/, 2016.

[66] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software plagiarism detection with birthmarks based on dynamic key instruction sequences," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1217–1235, 2015.

[67] L. Xue, C. Qian, and X. Luo, "Androidperf: A cross-layer profiling system for android applications," in *Proc. IWQoS*, 2015.

[68] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. ICSE*, 2017.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2017.2687880, IEEE Transactions on Information Forensics and Security

Ming Fan received his B.S. degree in computer science and technology from Xi'an Jiaotong University, China, in 2013. He is currently working toward the Ph.D. degree in the Department of Computer Science and Technology at Xi'an Jiaotong University, China. His research interests include trustworthy software and Android malware detection and classification.

Jun Liu received his B.S. and Ph.D. degrees in computer science and technology from Xi'an Jiaotong University in 1995 and 2004, respectively. He is currently a professor of the Department of Computer Science and Technology at Xi'an Jiaotong University in China. His current research focuses on data mining and text mining and he has authored more than seventy research papers in various journals and conference proceedings. He 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.

Wei Wang received his Ph.D. degree in control science and engineering from Xi'an Jiaotong University, China, in 2006. He is currently an associate professor in the School of Computer and Information Technology, Beijing Jiaotong University, China. He was a postdoctoral researcher in University of Trento, Italy, from 2005 to 2006. He was a postdoctoral researcher in TELECOM Bretagne and in INRIA, France, from 2007 to 2008. He was a European ERCIM Fellow in Norwegian University of Science and Technology (NTNU), Norway, and in Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, from 2009 to 2011. He visited INRIA, ETH, NTNU, CNR, and New York University Polytechnic. He is young AE of Frontiers of Computer Science Journal. He has authored or co-authored over 50 peer-reviewed papers in various journals and international conferences. His main research interests include mobile, computer and network security.

Haifei Li received his B.S. degree from Xi'an Jiaotong University, Xi'an, China, in 1990, and the M.S. and Ph.D. degrees from the University of Florida, Gainesville, FL, USA, in 1998 and 2001, respectively, all in computer science. He is currently an associate professor of Computer Science at Union University, Jackson, TN, USA. He has balanced academic / industrial experiences. He was an assistant professor of Computer Science at Nyack College from 2003 to 2004. He was a post-doctoral researcher at IBM Thomas J. Watson Research Center from 2001 to 2003. He was a graduate student at the University of Florida from 1996 to 2001. He was a software engineer at China Resources Information Technology from 1994 to 1996. He was a software engineer at Geophysical Research Institute from 1990 to 1994. He has published over 30 articles in various journals and magazines. His research interests are e-learning, database, e-commerce, automated business negotiation and business process management.

Zhenzhou Tian received his B.S. degree and Ph.D. degree in computer science and technology from Xi'an Jiaotong University, China, in 2010 and 2016, respectively. He is currently a lecturer in the School of Computer Science and Technology at Xi'an University of Posts and Telecommunications. His research interests include trustworthy software, software plagiarism detection, and software behavior analysis.

Ting Liu received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xi'an Jiaotong University, Xi'an, China, in 2003 and 2010, respectively. He is currently an associate professor of the Systems Engineering Institute, Xi'an Jiaotong University. His research interests include Smart Grid, network security, and trustworthy software.