IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received May 26, 2020, accepted June 3, 2020, date of publication June 8, 2020, date of current version June 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3000523

EstiDroid: Estimate API Calls of Android Applications Using Static Analysis Technology

WENHAO FAN, (Member, IEEE), DAISHUAI ZHANG, YE CHEN, FAN WU, (Member, IEEE), AND YUAN'AN LIU, (Member, IEEE)

Beijing Key Laboratory of Work Safety Intelligent Monitoring, School of Electronics Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding authors: Wenhao Fan (whfan@bupt.edu.cn) and Yuan'an Liu (yuliu@bupt.edu.cn)

This work was supported in part by the National Natural Science Foundations of China under Grant 61821001, in part by the YangFan Innovative and Entrepreneurial Research Team Project of Guangdong Province, and in part by the Fundamental Research Funds for the Central Universities.

ABSTRACT Tracking API calls of an Android application (app) has significant value for deeply understanding the app's running behaviors, so that to detect security damages, sensitive information leakages, energy consumptions, system resources occupations of the app, etc. However, existing methods track API calls of a target app through launching and manipulating the app in a real or simulated operating environment. The entire process is time consuming, which leads to low efficiency for practical system executing batch analysis for a considerable scale of apps. In order to enhance the speed of API calls tracking, in this paper, we propose a static analysis method, called EstiDroid, to estimate API calls of Android apps by statically analyzing the apps without actually running them. EstiDroid is composed of a static analyzer and an estimation algorithm. To analyze a target app, EstiDroid first obtains several types of static information from the app's .APK file via the static analyzer, then, the estimation algorithm is employed to establish the estimation model for the app based on the static information. Finally, according to the model, the proportion of each API's calls in the total number of calls is estimated. In experiments, 300 apps are tested via EstiDroid and manual operation in smartphone, the results show that EstiDroid only consumed 49242ms on average compared with manual testing, and it reached 84.06% average similarity and 90.74% maximum similarity compared with the API calls tracked in real environments.

INDEX TERMS Android, API calls tracking, static analysis, application behavior, smartphone.

I. INTRODUCTION

Android has become the most widely used mobile operating system (OS) for smartphones. The market share of Android in smartphone markets has reached at 85.1% (2018), and is still growing [1]. There are millions of Android applications (apps) developed and published in Android app markets. The number of available apps in the Google Play Store was most recently placed at 2.7 million apps in July 2019 [2]. However, the prosperity of Android apps also brings a series of challenges, such as security damages of malicious apps, sensitive information leakages of normal apps, excessive energy consumption of low-quality apps, intentional system resources occupations of rascal apps, etc. Thus, analyzing an Android app before delivering it to public is a necessity for Android app managements in order to ensure benignity and optimality of the app.

Currently, static and dynamic methods are two mainstream technologies for Android app analysis. The formers mainly adopt Android Virtual Machine bytecode analysis technology, which tries to convert .APK files of a target app into some intermediate representations, then generate the app's control flow graph, which reveals static information flows of the app. Whereas, the latters conduct real-time tracking, which actually runs and monitors the app in a smartphone or an Android simulator. The information flows are tracked during the running period of the app.

The Android APIs, written in Java language, act as interfaces used by Android apps to communicate with Android Framework. These APIs are vital to Android apps since they

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

provide apps system resources, information communications and life maintenance.

Tracking API calls of an Android app has significant value for deeply understanding the target app's running behaviors. API calls can be tracked by the dynamic methods through launching and manipulating the app on an Android smartphone or a simulator. However, the entire process is time consuming, which leads to low efficiency for practical system executing batch analysis for a considerable scale of apps; the static method analyzes the .APK file of the target app. The process is fast, but the result can only shows the app's control flow which reveals the relations among API calls, but it can not give any information about the number of API calls, which reflects the frequency that each API appearing in the execution flows of the app.

Aiming at boosting the speed of API calls tracking, in this paper, we propose an analysis method, called EstiDroid, to estimate API calls of Android apps by statically analyzing the apps without actually running them. It's an approach to high-speed API calls tracking through estimation based on static analysis technology.

EstiDroid consists of a static analyzer and an estimation algorithm. To analyze a target app, **(a)** the static analyzer is used to obtain several types of static information from the app's .APK file, including page layouts, manifest and intermediate representation; **(b)** the estimation algorithm is employed to establish the estimation model for the app based on the static information. Establishing the estimation model includes constructing entity description models, composing entity relationship graph and computing access intensities of entities. Then, the estimation algorithm estimates the proportion of each API's calls in the total number of calls, through traversing all entities in the entity relationship graph.

Experiments are conducted to evaluate the performance of EstiDroid. We picked up 00 apps from Android markets, then manually ran each of them on smartphones. API calls generated in the running period of each app were tracked using DroidInjector [3], a pre-installed dynamic API calls tracking tool that can track API calls during the running period of the app without modifying the Android OS. Then, we employed EstiDroid to estimate the API calls of these apps. It can be found that the estimated API calls via EstiDroid reached 84.06% similarity on average, 90.74% similarity on maximum (vs. 48 hours manual testing), in comparison with the tracked API calls via manual testing, whereas, EstiDroid only consumed 49242ms on average. The experiment results demonstrate the high efficiency of EstiDroid on estimating API calls of Android apps.

The rest of this paper is organized as follows: Section 2 discusses the related works. Section 3 presents the architecture of EstiDroid, including descriptions of the static analyzer and the estimation algorithm. Section 4 shows the experiment results of EstiDroid, and comparisons with manual testing and automatic testing. Section 5 concludes our work and introduces our future work briefly.

II. RELATED WORKS

In recent years, several representative Android app analysis technologies have been proposed, as are aforementioned, they are categorized into two types: static and dynamic methods.

A. STATIC METHODS

Android Virtual Machine bytecode analysis, which includes control-flow and data-flow analysis, is the main technology used by static methods. Control-flow analysis can help identify possible execution paths of the target app. Data-flow analysis can help predicate possible values of variables at some location of execution of the target app. In order to facilitate deep analysis, an intra-procedural or inter-procedural flow graph can be generated. FlowDroid [4] provides precise static tracking through parsing the converted intermediate representations of the target app. Android component lifecycle is modeled according to the call graph, which is attached with multiple dummy methods to identify lifecycle phases. ComDroid [5], AmanDroid [6], R-Droid [7], IccTA [8], DroidRA [9] and HornDroid [10] try to improve the static analyzer to detect implicit data flows across components among Android apps. LeakMiner [11] extracts Java byte code and metadata from the .APK file of the target app for processing, based on which, the call graph is then generated. LeakMiner is less context-aware since it does not mark lifecycle phases, so it may cause low precision and false positives. TrustDroid [12] carries out a detailed data flow tracking by converting Java byte code to tree structure, and then generating the call graph of the target app. TrustDroid can run on either a sever or a smartphone, but unfortunately, Android component lifecycle is not considered as well. Androguard [13] is an open-source, static analysis tool, can disassemble and decompile Android apps to make reverse engineering. Androguard unique Normalized Compression Distance approach can find similarities and differences in code between two apps, which can be used to detect repackaging. DroidMOSS [14] is a prototype detecting app repackaging using semantic file features. It extracts DEX opcode sequence from a target app, then generates a signature from it using fuzzy hashing technology. However, all above three systems have to treat libraries as black boxes since it is very hard to decompile their source codes. Tracking for inter-component communications is missing as well.

B. DYNAMIC METHODS

Bouncer [15] is a virtual machine based on dynamic analysis platform, which is used by Google officially to assess the security problems of apps uploaded by third-party developers. Bouncer runs app to check any malicious behaviors and compares them with previous analyzed malicious apps. TaintDroid [16] is a system-level dynamic tracking system for Android. Sensitive information is tagged for being tracked from tainted sources to sinks. The target app under analysis is executed in emulated environment to perform taint-analysis and API monitoring. Many systems [17]–[21] are based on

TaintDroid to conduct further analysis. Kynoid [22] is based on TaintDroid, and it implements a middleware between app and data in Android system to provide a runtime security policy enforcement for app accessing shared data. Andromaly [23] is a light-weight dynamic analysis tool which performs real-time monitoring for collection of various system metrics, including CPU usage, amount of network data, number of active processes and battery usage, etc. Although, Andromaly can not monitor API calls during the running period of target app. DroidTrace [24] proposed an implementation of a ptrace-based dynamic tracking system, which can monitor selected Linux system calls invoked during the running period of the target app. DroidInjector [3] is a dynamic tracking tool which can monitor API calls in Android Virtual Machine Runtime, which can provide more fine-grained analysis results compared with Linux system calls monitoring. It uses multiple technologies, such as Linux ptrace, JNI conversion, etc., to execute context-aware, flow-aware and library-aware API calls tracking for the target app.

C. METHODS FOR TRACKING API CALLS

Dynamic methods provide real or simulated environments where apps can be installed, executed and operated. The Android OS in the environments is modified, so that an API call will be tracked once the target app calls the API. The time consumption of the entire process is high, because most of time is consumed in the process of operating the app, where user/system inputs are carried out through Graphical User Interface (GUI) operations and system event triggers. A human tester has to manually operate the app for a time period long enough to ensure most of possible inputs for the app are executed. For an automatic test, test tools like Robotium [25], Monkeyrunner [26] actually replace human tester to carry out the inputs of the target app. However, the process still takes a long time since the test tools change the input patterns merely, but running and operating the app remain unchanged.

Existing static methods can give the target app's control flow which only reveals the relations among API calls. These methods are fast since the result is generated by analyzing the .APK file of the app without actually running the app, but they are not capable to provide any information about the number of API calls. The number of calls for a certain API essentially reflexes the frequency of the API appearing in the execution flow of the app. It is an important criteria in Android app analysis. For example, we can evaluate security problems of the target app by observing abnormal number of API calls. If the energy consumption of each API is obtained previously, we can also predicate energy consumptions of the target app according to the frequency of each API.

III. ARCHITECTURE OF EstiDroid

How to obtain API calls of Android apps, especially the number of calls of each API, through an efficient way which only consumes a very short time? EstiDroid is an approach to high-speed API calls tracking through estimation based on

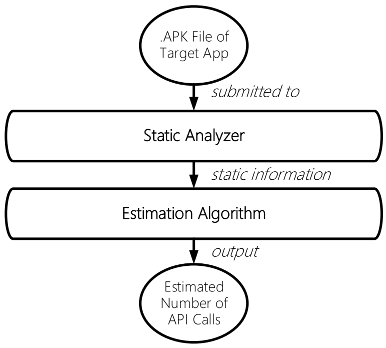


FIGURE 1. Steps of estimating API calls.

static analysis technology. EstiDroid doesn't need a smartphone environment or a simulator environment to operate the target app, whereas, it uses static analysis technology to obtain several types of static information from the .APK file of the app, and then estimates API calls according the estimation model established based on the static information. Thus, tracking API calls via EstiDroid is rather faster compared with dynamic methods, meanwhile, the API calls estimated by EstiDroid keep high similarity with those tracked when the app runs in actual environment.

In this section, the architecture of EstiDroid is presented. Firstly, we describe the steps used by EstiDroid to estimate API calls of a target Android app. Then, we explain in detail about the two components of EstiDroid: static analyzer and estimation algorithm.

A. STEPS OF ESTIMATING API CALLS

EstiDroid contains two components: a static analyzer and an estimation algorithm. The output from the static analyzer is used by the estimation algorithm.

As shown in Fig. 1, there are mainly two steps to analyze a target app:

1) The static analyzer carries out XML file parsing and Android Virtual Machine bytecode analysis for the .APK file of the target app. Extracted XML files and converted code files generated by the static analyzer is then used to output several types of static information, including page layouts, manifest and intermediate representation. Properties of widgets, components and entry functions, and relations among API calls of the app can be obtained from the static information.

2) The estimation algorithm establishes the estimation model for the target app through constructing entity description models, composing entity relationship graph, computing access intensities of entities and estimating the proportion of each API's calls in the total number of calls. The entity description model of an entity is a structure which contains necessary attributes of the entity. The entity relationship graph, which consists of entity description models, expresses relations among the entities. The access intensity of an entity is the weight reflecting the probability of the entity being accessed by the execution flow of the app. The execution flow

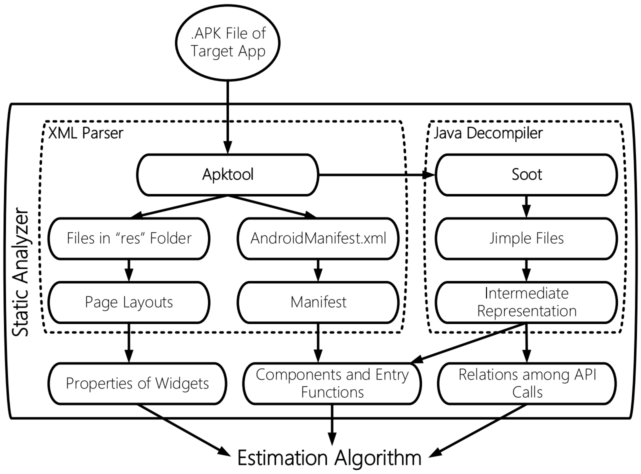


FIGURE 2. **Structure of static analyzer.**

is impacted by the user's and system's common input. To estimate API calls, the estimation algorithm traverses all entities in the entity relationship graph. The result of estimated API calls can be computed according to access intensities during the traversing. When the traversing ends, the estimation is completed.

B. STATIC ANALYZER

The static analyzer is composed of a XML parser and a Java decompiler, as shown in Fig. 2.

1) XML PARSER

The XML parser is implemented based on Apktool [27], a tool for reverse engineering of Android .APK files. The XML parser uses Apktool to extract AndroidManifest.xml and layout files in 'res' folder from the .APK file of the target app.

Android components used in the app are declared in AndroidManifest.xml, thus, the XML parser can get the name list of these components including **Activity**s, **Service**s, **Application**s and statically registered **BroadcastReceiver**s. The XML parser can find the start point of the app through analyzing 'intent-filter' labels defined in AndroidManifest.xml, and it can also find the **Service**s that can be launched remotely through analyzing the tag 'android:process = ":remote"' in the declaration of each **Service** in AndroidManifest.xml.

The page layout for each activity in the app is declared in xml files in the 'res' folder of the app. The XML parser first reads xml files in the 'value' sub-folder, in order to get 'name-ID' correspondence for each layout and widget. Then, the XML parser obtains necessary properties, including type, location and size, of the widgets in the page of each **Activity** through traversing these xml layout files.

2) JAVA DECOMPILER

The Java decompiler is implemented based on Soot [28]. Soot is a code analysis tool which converts the Android Virtual Machine bytecode of the target app into an intermediate representation called Jimple [28]. The java decompiler uses Soot after the app's .APK file being extracted by Apktool.

Firstly, the Java decompiler traverses Jimple files, and finds out Jimple files corresponding to each **Activity**, **Service** and **Application** in the name list obtained by the XML parser.

Then, the Java decompiler traverse the contents in these files, and **(a)** finds all lifecycle functions for each **Activity**, **Service** and **Application** via scanning key words, such as 'onCreate', 'onResume', 'onStop', etc.; **(b)** finds all **BroadcastReceiver**s (including statically registered **BroadcastReceiver**s), accepted broadcasts (that is, names of the broadcasts) by scaning the keyword 'BroadcastReceiver' and corresponding listener functions like 'setXXXListener'; **(c)** finds each widget and its event handlers in each **Activity** through scanning the function 'setContentView' and 'findViewById', referring the correspondence between the widget's name and ID, and marking listener functions like 'setXXXListener'.

Additionally, in part **(a)** - **(c)**, the Java decompiler also establishes execution flow graph for every entry function, including lifecycle functions of **Activity**s, **Service**s and **Application**s, and event handler functions of **BroadcastReceiver**s and widgets. The execution flow graph for an entry function is a representation, using graph notation, of all paths that might be traversed starting from the entry function during the app's execution. The APIs called in each path are marked, start points and end points of loops using '**for**', '**while**', and '**do while**' are marked, and start points and end points of branches in conditional judgments using '**if-else**' and '**switch-case**' are also marked.

The flowchart of the static analyzer is shown in Fig. 3. Finally, properties of widgets, components and entry functions, and relations among API calls are all obtained through running the static analyzer. The above static information is then delivered to the estimation algorithm.

The whole process of the static analyzer analyzing a target app is very fast since the time is mainly consumed by extracting the .APK file, converting original code into Jimple representation, traversing the Jimple files and scanning texts in these files. All of them only require computing resources. Thus, the speed of the process can be further boosted if a more powerful CPU is employed.

C. ESTIMATION ALGORITHM

The estimation algorithm exploits the static information generated by the static analyzer to estimate API calls for the target app.

As the flowchart shown in Fig. 4, the process of the estimation algorithm consists of 4 steps: constructing entity description models, composing entity relationship graph, computing access intensities, and estimating API calls.

1) CONSTRUCT ENTITY DESCRIPTION MODELS

We employ the term entity to uniformly describe Android structures used in the estimation algorithm, including

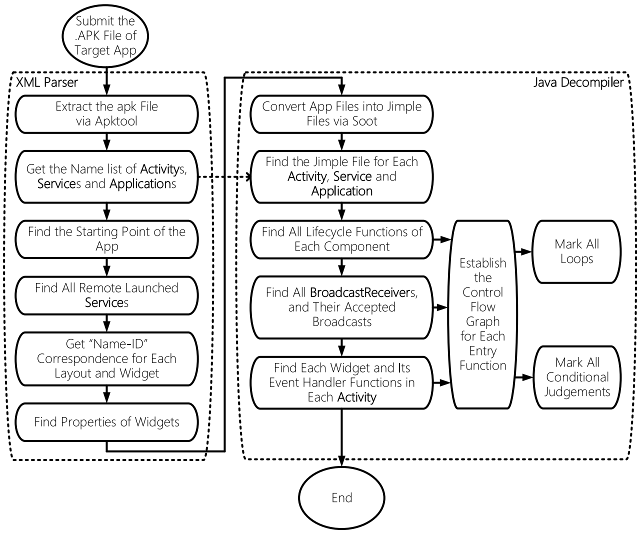


FIGURE 3. Flowchart of static analyzer.

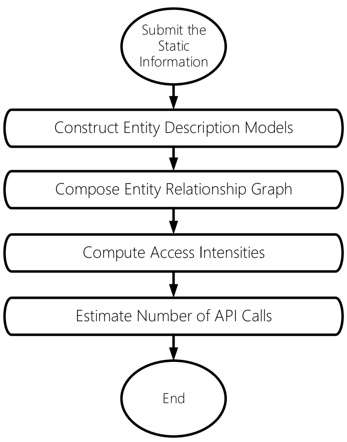


FIGURE 4. Flowchart of the estimation algorithm.

TABLE 1. Entity description models.

Activity a	
$.name$	**Activity**'s name
$.\eta$	access intensity of the **Activity**
$.\mathcal{W}$	$= \{w, \ldots\}$, set of widgets in page layout of the **Activity**
$.\mathcal{E}$	$= \{e, \ldots\}$, set of all rewritten lifecycle functions in the **Activity**
$.\mathcal{A}$	$= \{a, \ldots\}$, set of all successive **Activitys**
$.\mathcal{S}$	$= \{s, \ldots\}$, set of all successive **Services**
$.\mathcal{B}$	$= \{b, \ldots\}$, names of all broadcasts that the **Activity** can send
Application p	
$.name$	**Application**'s name
$.\mathcal{E}$	$= \{e, \ldots\}$, set of all rewritten lifecycle functions in the **Application**
Service s	
$.name$	**Service**'s name
$.\eta$	access intensity of the **Service**
$.type$	$= remote$ if the **Service** can be launched remotely, or $= local$ if the **Service** can be launched locally only
$.\mathcal{E}$	$= \{e, \ldots\}$, set of all rewritten lifecycle functions in the **Service**
BroadcastRecevier r	
$.name$	**BroadcastReceiver**'s name
$.\eta$	access intensity of the **BroadcastReceiver**
$.\mathcal{B}$	$= \{b, \ldots\}$, names of all broadcasts accepted of the **BroadcastReceiver**
$.e$	event handler function in the **BroadcastReceiver**
Widget w	
$.name$	widget's name
$.\eta$	access intensity of the widget
$.type$	widget's type
$.size$	widget's size
$.location$	widget's location
$.\mathcal{E}$	$= \{e, \ldots\}$, set of all event handler functions in the widget
$.\mathcal{A}$	$= \{a, \ldots\}$, set of all successive **Activity**s and the entry that operates the jump
$.\mathcal{S}$	$= \{s, \ldots\}$, set of all successive **Service**s and the entry that operates the launch
$.\mathcal{B}$	$= \{b, \ldots\}$, names of all broadcasts that the widget can send
Entry e	
$.name$	entry function's name
$.\mathcal{F}$	$= \{f, \ldots\}$, set of all APIs in the entry function
API f	
$.name$	API's name
$.q$	API's frequency in the execution flow graph of the entry function

Activity, **Application**, **Service**, **BroadcastReceiver**, widget, entry function and API. The entity description model of an entity is a collection which contains the entity's attributes needed by the estimation algorithm.

Entity description models of different entities are described in TABLE 1, where attributes and their corresponding explanations are illustrated as well.

1.1) Entity Description Model for an **Activity**

The entity description model for an **Activity** a contains 7 attributes: *name*, η, $\mathcal{W}$, $\mathcal{E}$, $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{B}$.

1.2) Entity Description Model for an **Application**

The entity description model for an **Application** p contains 2 attributes: *name* and $\mathcal{E}$.

1.3) Entity Description Model for a **Service**

The entity description model for a **Service** s contains 4 attributes: *name*, η, *type* and $\mathcal{E}$. Note that, the value of *type* identifies whether the **Service** can be launched remotely, or locally only.

1.4) Entity Description Model for a **BroadcastReceiver**

The entity description model for an **BroadcastReceiver** r contains 4 attributes: *name*, η, *broadcasts* and e.

1.5) Entity Description Model for a Widget

The entity description model for a widget contains 10 attributes: *name*, η, *type*, *size*, *location*, $\mathcal{E}$, $\mathcal{W}$, $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{B}$. Note that, *type* expresses the type that current widget belongs to, such as *Button*, *ListView*, *EditView*, *CheckBox*, *ImageView*, etc. *size* denotes the size (height×width) of current widget in the page layout. *location* express the location of current widget in the page layout. As shown in Fig. 5, the value of *location* is selected from {*left_top*, *middle_top*, *right_top*, *center*, *bottom*}. The location of a widget is decided by its relative location compared with other widgets in the page layout.

Note that, in 1.1) - 1.5), an element a in $\mathcal{A}$ represents an successive **Activity** that current entity can jump to; an element s in $\mathcal{S}$ represents a successive **Service** that current entity can launch; an element b in $\mathcal{B}$ represents a broadcast that current entity can send.

1.6) Entity Description Model for an Entry Function

The entity description model for an entry function contains 2 attributes: *name* and $\mathcal{F}$. Note that, the set $\mathcal{F}$ contains all APIs called in the execution flow graph (generated by the

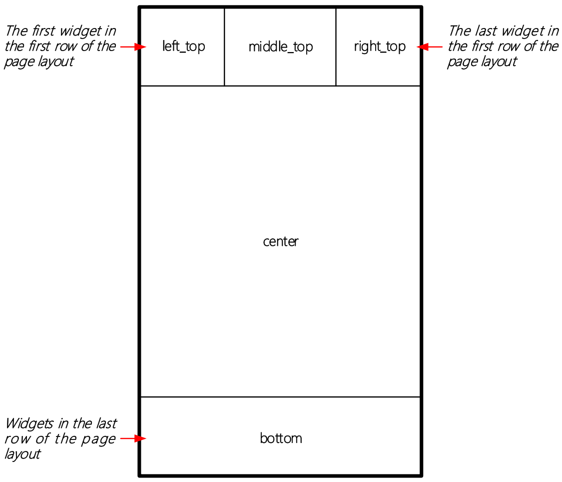


FIGURE 5. Value of *w.location* for a widget in a page layout.

Java decompiler of the static analyzer) starting from the entry function.

1.7) Entity Description Model for an API

The entity description model for an API contains 2 attributes: *name* and *q*.

We use frequency to express the frequency of an API appearing in the execution flow graph (generated by the Java decompiler of the static analyzer), namely, the frequency of calls for the API in the execution flow when the entry function is called once.

For an API f, if it is in a loop, $f.q$ will increase by the count number of the loop, since f will be called multiple times in the loop. If the count number of the loop is unknown, we will assign a constant value Ω for simplicity. If f is in a branch of a conditional judgement, $f.q$ will increase based on the number of branches of the conditional judgement. In the running period of the app, the API will be called probabilistically because only one branch in all branches is called by the execution flow. For simplicity, we consider it follows average probability distribution among all branches, so $f.q$ will increase by $1/(\text{number of branches})$. Note that, for an API in nested loops, nested conditional judgements, or combined loops and conditional judgements, the total increased frequency is the product of increased frequency generated at each loop or conditional judgement in the nested or combined structure.

Configuring each API in the $\mathcal{F}$ of an entry function e is completed through traversing the execution flow graph (generated by the Java decompiler of the static analyzer) of e.

The detailed process of is given in **Algorithm 1**.

In **Algorithm 1**, we denote δ as frequency cache, which temporally caches the value of frequency for current API. When the algorithm enters into a loop or a branch of a condition judgement, δ will accumulate the increased frequency generated by the loop or branch; when the algorithm exits from the loop or branch, δ will wipe off the increased frequency correspondingly. When the algorithm meets an API, δ will be added to frequency of the API.

Algorithm 1 Process of Configuring Each *api* in the *apis* an Entry Function e_n

Input: e_n and its execution flow graph
Output: $e_n.\mathcal{F}$
$\delta <= 1$; // frequency cache
for each node $node_i$ in the execution flow graph of e_n **do**
  if $node_i$ is an API **then**
    if the API doesn't exist in $e_n.\mathcal{F}$ **then**
      create an entity f_j for current API;
      $f_j.name <=$ current API's name;
      $f_j.r <= \delta$;
      add f_j into $e_n.\mathcal{F}$;
    else //f_j already exists in $e_n.\mathcal{F}$
      $f_j.r <= f_j.r + \delta$;
    end if
  else if $node_i$ is the start of a loop **then**
    $\delta <= \delta \cdot$ (count number of the loop) or $\delta \cdot \Omega$ if the count number is unknown;
  else if $node_i$ is the end of a loop **then**
    $\delta <= \delta/$(count number of the loop) or δ/Ω if the count number is unknown;
  else if $node_i$ is the start of a branch of a conditional judgement **then**
    $\delta <= \delta/$number of branches;
  else if $node_i$ is the end of a branch of a conditional judgement **then**
    $\delta <= \delta \cdot$ number of branches;
  end if
end for

In order to construct entity description models, the estimation algorithm scans the static information generated by the static analyzer, creates the entity description model for each entity, assigns *name* for all entities, assigns *type*, *size* and *location* for all widgets, assigns *type* for all **Service**s, and assigns accepted broadcasts into $\mathcal{B}$ for all **BroadcastReceiver**s. Besides, the estimation algorithm builds the affiliations between each **Activity/Application/Service** and its entry functions, between each **BroadcastReceiver** and its entry function, between each **Activity** and its widgets, and between each entry functions and its APIs.

The detailed process is given in **Algorithm 2**.

2) COMPOSE ENTITY RELATIONSHIP TREE

The entity relationship graph is a graph-like structure where nodes are composed of entities description models of the app, and a link between two nodes represent the relation between two corresponding entities, including jumps of **Activity**s, launchings of **Service**s and sending of broadcasts. For example, a link from **Activity** a_1 to a_2 means a_1 can jump to a_2; a link from a_1 to **Service** s_1 means a_1 can launch s_1; a link from a_1 to broadcast b_1 means a_1 can send b_1;

In order to compose the entity relationship graph for the app, the estimation algorithm scans the entity description models and the static information generated by the static

Algorithm 2 Process of Constructing Entity Description Models

Input: static information generated from the static analyzer
Output: entity description models
for each **Activity**, **Application**, **Service**, **BroadcastReceiver**, widget and entry function **do**
  create the entity description model for current entity;
  name <= the entity's name;
  if the entity is a widget **then**
    type <= the type of the widget;
    size <= the proportional size of the widget;
    location <= the relative position of the widget;
    add current widget into $\mathcal{W}$ in the entity of **Activity** it belongs to;
  else if the entity is a **Service then**
    type <= the **Service**'s type, which is identified by XML Parser;
  else if the entity is a **BroadcastReceiver then**
    add the names of all broadcasts accepted by the **BroadcastReceiver** into $\mathcal{B}$;
  else if the entity is an entry function **then**
    configure $\mathcal{F}$ according to **Algorithm 2**;
    add current entry function into $\mathcal{E}$ (for **Activity**, **Application**, **Service** and widget) or e (for **BroadcastReceiver**) in the entity that the entry function belongs to;
  end if
end for

analyzer. It links relevant models by adding elements into the set $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{B}$.

The detailed process is given in **Algorithm 3**.

In Android, jumping to an **Activity** is completed by calling API 'startActivity' or 'startActivityForResult', where the destination **Activity** is designated in the parameters. So we can find relations of **Activity** jumps through scanning the above two functions and their parameters. Similarly, launching a **Service** relies on calling API 'startService' or 'bindService', and sending a broadcast relies on calling 'sendBroadcast' or 'sendOrderedBroadcast'. The relations of **Service** launching, and relations between broadcasters and **BroadcastReceiver**s can be found by scanning these APIs and their parameters.

3) COMPUTE ACCESS INTENSITIES

The access intensity of an entity, including widget, **Activity**, **Service**, **BroadcastReceiver**, denotes its weight which measures the probability of the entity being accessed by the execution flow caused by the app's common input.

The common input of an app represents the user's normal operations and the system's normal event triggers during the running period of the app. It reflects the statistical result of common user behaviors, and system activities. If an entity has high access intensity, it means the execution flow of the common input accesses the entity with high probability, that is, APIs in the entity will be called more frequently; if an entity has low access intensity, it means the execution flow of the common input accesses the entity with low probability, that is, APIs in the entity will be called more occasionally.

Algorithm 3 Process of Composing Entity Relationship Graph

Input: entity description models, static information generated by the static analyzer
Output: entity relationship graph
for each entity description model of **Activity**s and widgets **do**
  for each entry function in $\mathcal{E}$ or e of current entity **do**
    for each API f_i in $\mathcal{F}$ **do**
      if $f_i.name == startActivity$ **or** $f_i.name == startActivityForResult$ **then**
        add the **Activity** designated in the parameters of f_i into $\mathcal{A}$ of current entity;
      else if $f_i.name == startService$ **or** $f_i.name == bindService$ **then**
        add the **Service** designated in the parameters of f_i into $\mathcal{S}$ of current entity;
      else if $f_i.name == sendBroadcast$ **or** $f_i.name == sendOrderedBroadcast$ **then**
        add the name of the broadcast designated in the parameters of f_i into $\mathcal{B}$ of current entity;
      end if
    end for
  end for
end for

TABLE 2. Weight factor of each type of widget.

Type of Widget	Weight Factor
$Button, ViewButton, ...$	2.5
$ListView$	3
$EditView$	1.5
$CheckBox$	2
$ImageView$	1.5
...	...

a: ACCESS INTENSITY OF WIDGET

Considering the use habit of users, we design that the access intensity of a widget is decided by its type, location and size. A user is more likely to operate a widget with specified type, hotspot location, and big size, thus, such widget should have higher value of access intensity, else it should be assigned as a low value.

i) TYPE OF WIDGET

We consider the fact that type of widget impacts the frequency of user's operations on it. For example, *Button* widgets get more clicks of users than *ImageView* widgets in user's common usage. We statically allocate a weight factor for each type of widget, as shown in TABLE 2 partially. We express as $t_k^{(t)}$ the weight factor of widget w_k's type.

TABLE 3. Weight factor of each location of widget.

Location of Widget	Weight Factor
left_top	0.5
middle_top	0.25
right_top	0.75
center	1.5
bottom	2

ii) LOCATION OF WIDGET

As shown in Fig. 5, more specifically, on the layout of the app, the widget at *left_top* is always in charge of returning to last page; the widgets at *middle_top* are basically the label of current page or search bar, etc.; the widget at *right_top* can be menus, settings or other functionalities, etc.; the widgets at *center* are accessed by most of activities in current page; user frequently-used widgets are deployed at *bottom* since the area are the nearest to user's fingers. We set a weight factor for each location of widget, as shown in TABLE 3. We express as $t_k^{(l)}$ the weight factor of widget w_k's location.

iii) SIZE OF WIDGET

The size of a widget impacts the popularity of the widget being accessed by the user. Generally, the larger the widget is, the more attractive the widget will become, so the more frequent the user will access the widget. We express the size of widget w_k as $t_k^{(s)}$, the value of which is the area of w_k, that is, height $\times$ width.

iv) COMPUTATION OF $w_k.\eta$

For widget w_k in **Activity** a_i, its intensity $w_k.\eta$ is computed by averaging the normalized $t_k^{(t)}$, $t_k^{(l)}$ and $t_k^{(s)}$. Thus, $w_k.\eta$ can be given by

$$\begin{aligned} w_k.\eta &= \alpha \bar{t}_k^{(t)} + \beta \bar{t}_k^{(l)} + \gamma \bar{t}_k^{(s)} \\ &= \alpha \frac{t_k^{(t)}}{\sum_{w_j \in a_i.\mathcal{W}} t_j^{(t)}} + \beta \frac{t_k^{(l)}}{\sum_{w_j \in a_i.\mathcal{W}} t_j^{(l)}} + \gamma \frac{t_k^{(s)}}{\sum_{w_j \in a_i.\mathcal{W}} t_j^{(s)}} \end{aligned} \tag{1}$$

where $\sum_{w_j \in a_i.\mathcal{W}} t_j^{(t)}$ is the sum of all weight factors of types of all widgets in a_i, so do $\sum_{w_j \in a_i.\mathcal{W}} t_j^{(l)}$ and $\sum_{w_j \in a_i.\mathcal{W}} t_j^{(s)}$ correspondingly. α, β and γ are balance factors to tune the proportion of the weight of type, location and size. $\alpha, \beta, \gamma \in [0, 1]$ and $\alpha + \beta + \gamma = 1$.

The computation of access intensities of widgets is done by traversing each widget of each **Activity** in entity relationship graph. The process is described in **Algorithm 4**.

Algorithm 4 Process of Computing Access Intensities of Widgets

Input: entity relationship graph
Output: entity relationship graph where η of all widgets are computed
for each **Activity** a_i in entity relationship graph **do**
  for each widget w_k in $a_i.\mathcal{W}$
    compute $t_k^{(t)}$ according to $w_k.type$;
    compute $t_k^{(l)}$ according to $w_k.location$;
    compute $t_k^{(s)}$ according to $w_k.size$;
    compute $w_k.\eta$ according to Formula (1);
  end for
end for

b: ACCESS INTENSITY OF ACTIVITY

In an Android app, an **Activity** manages a page of the app. When a user operates the app, a page can jump to another one according to the user's commands through clicking screen of the smartphone, pressing buttons of the smartphone, etc. The jumps among **Activity**s of the app is actually a network where **Activity**s are nodes, and links are jumps. A link connecting **Activity** a_1 and a_2 indicates that a_1 can jump to a_2, and a_2 can return to a_2 through return operation.

In order to abstract user operations on **Activity**s of the target app, we propose a PageRank-based algorithm to compute the access intensities of **Activity**s. The PageRank algorithm [29] is used by Google Search to rank websites in their search engine results, whereas, here we modify the PageRank algorithm to evaluate the importance of **Activity**s in the target app.

Our PageRank-based algorithm works by evaluating the jumps from or to an **Activity** to determine an estimation in regard to how important the **Activity** is. The underlying assumption is that more important **Activity**s are likely to have more jumps from other **Activity**s.

A jump from **Activity** a_1 to a_2 is triggered by calling the function '*startActivity*' or '*startActivityForResult*', which can be included in event handler functions of the widgets in a_1 and lifecycle functions of a_1. As aforementioned, all jumps starting from a_1 are recorded in $a_1.\mathcal{A}$ and all $\mathcal{A}$ of the widgets belonging to a_1. Additionally, if a_1 can jump to a_2, a_2 can also return to a_1 through user pressing 'back' button.

Thus, we have 3 groups of types of **Activity** jumps: **(a)** jumps via event handler functions of widgets; **(b)** jumps via returning from previous **Activity**s; **(c)** jumps via lifecycle functions of **Activity**s. The jumps in group **(a)** and **(b)** are triggered by user operations, whereas, the jumps in group **(c)** are triggered by the app automatically.

For a certain **Activity** a_i, we define 3 sets $jumps_i^{(w)}$, $jumps_i^{(r)}$ and $jumps_i^{(l)}$ for above 3 groups of jumps. Necessary information is collected into the 3 sets through traversing all jumps of the target app.

For each jump in group **(a)**, $jumps_i^{(w)}$ contains each widget that can jump to a_i and the **Activity** that the widget belongs to. The elements in $jumps_i^{(w)}$ are ordered pairs. For example, if in a jump, **Activity** a_1 can jump to a_i via its widget w_1, then we have $< w_1, a_1 > \in jumps_i^{(w)}$.

For each jump in group **(b)**, $jumps_i^{(r)}$ includes each destination **Activity** that a_i can jump to, and the total number of jumps from other **Activity**s to the destination **Activity**. For example, if in a jump, a_i can jump to a_1, and a_1 has c_1

jumps from other **Activity**s to it, then we have $< a_1, c_1 > \in jumps_i^{(r)}$.

For each jump in group **(c)**, $jumps_i^{(l)}$ consists of all **Activity**s that can jump to a_i via lifecycle functions. For example, if in a jump, a_1 can jump to a_i by calling *onCreate*, then we have $a_1 \in jumps_i^{(l)}$.

Therefore, we design that intensity $a_i.\eta$ is composed of $\eta_i^{(w)}$, $\eta_i^{(r)}$ and $\eta_i^{(l)}$, which are the intensities from above 3 groups, respectively. $a_i.\eta$ can be computed by

$$a_i.\eta = (1-\delta)\eta_i^{(w)} + \delta\eta_i^{(r)} + \eta_i^{(l)} \tag{2}$$

where δ $(0 < \delta < 1)$ is proportion factor which represents the proportion of user's return operations in all user's operations on a_i.

$\eta_i^{(w)}$ is computed through summing up the proportional intensities of all **Activity**s which can jump to a_i via their widgets. It can be given by

$$\eta_i^{(w)} = \sum_{<w_x,a_y>\in jumps_i^{(w)}} \left(\frac{w_x.\eta}{\sum_{w_z\in a_y.\mathcal{W}}(|w_z.\mathcal{A}|w_z.\eta)} a_y.\eta\right) \tag{3}$$

where $|w_z.\mathcal{A}|$ is the number of elements in $w_z.\mathcal{A}$, thus $w_x.\eta/(\sum_{w_z\in a_y.\mathcal{W}}(|w_z.\mathcal{A}|w_z.\eta))$ is the proportion of the intensity of widget w_x in the sum of intensities of a_y's widgets.

$\eta_i^{(r)}$ is computed through accumulating the averaged intensity of each **Activity** that a_i can jump to. It is given by

$$\eta_i^{(r)} = \sum_{<a_x,c_x>\in jumps_i^{(r)}} \left(\frac{1}{c_x} a_x.\eta\right) \tag{4}$$

where $1/c_x$ shows the probability of returning operations is averaged by all jumps to a_i.

$\eta_i^{(l)}$ is computed through accumulating the intensity of each **Activity** that can jump to a_i through lifecycle functions. It is given by

$$\eta_i^{(l)} = \sum_{a_x\in jumps_i^{(l)}} a_x.\eta \tag{5}$$

In order to guarantee the convergence of our PageRank-based algorithm, if an **Activity** has no jumps from itself to other **Activity**s, it will contribute all its intensity averagely to the **Activity**s that can jump to it according to Formula (4). The above rule is used to replace the random surfing rule for sink nodes in standard PageRank algorithm, since in Android, a user can not switch to an arbitrary page from current page.

Algorithm 5 gives the whole process to compute access intensities of **Activity**s. First, the algorithm initialize all **Activity**s, then the access intensity of each **Activity** is computed iteratively. In each iteration, the gap between the access intensities of each **Activity** in current iteration and previous iteration are measured. The access intensity of each **Activity** become stable with the increase of the number of iterations. The iteration will terminate when the sum of the gaps of all **Activity**s is less than or equal to ϵ ($gap \le \epsilon$), which means the convergence of the algorithm is considered to be reached. Finally, the access intensity of each **Activity** is normalized through computing the proportion of the access intensity of the **Activity** in the sum of access intensities of all **Activity**s.

Algorithm 5 Process of Computing Access Intensities of **Activity**s

Input: entity relationship graph
Output: entity relationship graph where η of all **Activity**s are computed

```
for each Activity a_i in entity relationship graph do
  a_i.η <= 1;  // set initial value as 1
  assign elements for jumps_i^(w), jumps_i^(r), jumps_i^(l);
end for
do
  for each Activity a_i in entity relationship graph do
    η_i^(prev) <= a_i.η;  // temporarily cache a_i.η computed in
                          // the previous iteration
    compute η_i^(w) according to Formula (3);
    compute η_i^(r) according to Formula (4);
    compute η_i^(l) according to Formula (5);
    compute a_i.η according to Formula (2);
    gap_i <= |a_i.η − η_i^(prev)|;  // compute the gap between a_i.η
                                    // computed in current and previous
                                    // iteration
  end for
  gap <= Σ gap_i;  // sum up the gaps of all Activitys
while (gap > ε)  // algorithm converges when gap ≤ ε
  η <= Σ a_i.η;  // sum up access intensities of all Activitys
for each Activity a_i in entity relationship graph do
  a_i.η <= a_i.η/η; // normalize the access intensity of each Activity
end for
```

c: ACCESS INTENSITY OF SERVICE

A **Service** in the target app can be launched from **(a)** inside of the app by calling function '*startService*' or '*bindService*' in the execution flow. We call it local launching; **(b)** outside of the app through the system or other apps sending system events. We call it remote launching. For a certain **Service**, its type is identified by *type* of the **Service**.

As aforementioned, $\mathcal{S}$ of each **Activity** and widget contains the **Service**s that current entity can launch, which belong to local launching.

For a **Service** s_l of the app, if $s_l.type == local$, which means s_l can be only launched locally, then we consider its access intensity $s_l.\eta$ inherits from the access intensities of all **Activity**s (including their widgets) that can launch the **Service**.

s_l can be remotely launched if $s_l.type == remote$. The access intensity $s_l.\eta$ consists of two parts: **(a)** access intensity inheriting from its launcher **Activity**s (including their widgets); **(b)** access intensity caused by remote launchings. In order to model remote launching, we define a total access intensity ρ representing the sum probability for all **Service**s launched remotely. Thus, s_l's access intensity caused by

Algorithm 6 Process of Computing Access Intensities of **Service**s

Input: entity relationship graph
Output: entity relationship graph where η of all **Service**s are computed
for each **Service** s_l in entity relationship graph **do**
  if $s_l.type == local$ **then**
    $s_l.\eta <= 0$; // set initial value as 0
  else // $s_l.type == remote$
    $s_l.\eta <= \rho/$(the number of all remotely launched **Service**s);
  end if
end for
for each **Activity** a_i in entity relationship graph **do**
  for each **Service** s_l in $a_i.\mathcal{S}$
    $s_l.\eta <= s_l.\eta + a_i.\eta$; // inherit the access intensity of the // launcher **Activity**
  end for
  for each widget w_k in $a_i.\mathcal{W}$
    for each **Service** s_l in $w_k.\mathcal{S}$
      $s_l.\eta <= s_l.\eta + a_i.\eta \cdot w_k.\eta$; // inherit the access intensity of // the launcher widget of current // **Activity**
    end for
  end for
end for

remote launchings is computed by averaging ρ by the number of all remotely launched **Service**s.

In summary, the whole process of computing access intensities of **Service**s is described in **Algorithm 6**.

d: ACCESS INTENSITY OF *BroadcastReceiver*

Similarly, a **BroadcastReceiver** in the target app can receive the broadcasts sent from **(a)** inside of the app by calling '*sendBroadcast*' and '*sendOrderedBroadcast*', we call it local broadcasting; **(b)** outside of the app through system or other apps sending broadcasts, we call it remote broadcasting.

As aforementioned, the $\mathcal{B}$ of each **Activity** and widget contains the broadcasts that current entity can send, which belong to local broadcasting.

Remote broadcasting mainly comes from system broadcasts, such as SCREEN_ON_ACTION, SCREEN_OFF_ACTION, CALL_ACTION, ANSWER_ACTION, DATA_CONNECTION_STATE_CHANGED_ACTION, SERVICE_STATE_CHANGED_ACTION, SIGNAL_STRENGTH_CHANGED_ACTION, etc. If a **BroadcastReceiver** in the app receives system broadcasts, its $\mathcal{B}$ has the names of corresponding system broadcasts. We define as τ the total probability of the generation of system broadcasts.

Thus, for a **BroadcastReceiver** r_m, its access intensity $r_m.\eta$ includes two parts: **(a)** local broadcasting: the access intensity inherited from the access intensities of **Activity**s and widgets which can send broadcasts to r_m; **(b)** remote broadcasting: τ multiplied by the access intensity brought by the access intensities of the system broadcasts that r_m receives. We define the access intensity of each system broadcast to represent the probability that the system generates the broadcast, as shown in TABLE 4 partially.

TABLE 4. Access intensity of each system broadcast.

System Broadcast	Access Intensity
SCREEN_ON_ACTION	25%
SCREEN_OFF_ACTION	25%
CALL_ACTION	5%
ANSWER_ACTION	5%
DATA_CONNECTION_STATE _CHANGED_ACTION	8%
...	...

Algorithm 7 Process of Computing Access Intensities of **BroadcastReceiver**s

Input: entity relationship graph
Output: entity relationship graph where η of all **BroadcastReceiver**s are computed
for each **BroadcastReceiver** r_m in entity relationship graph **do**
  $r_m.\eta <= 0$; // set initial value as 0
end for
for each **Activity** a_i in entity relationship graph **do**
  for each broadcast b_n in $a_i.\mathcal{B}$
    for each **BroadcastReceiver** r_m whose $r_m.\mathcal{B}$ contains b_n **do**
      $r_m.\eta <= r_m.\eta + a_i.\eta$; // inherit the access intensity of the // **Activity** that sends b_n
    end for
  end for
  for each widget w_k in $a_i.\mathcal{W}$
    for each broadcast b_n in $w_k.\mathcal{B}$
      for each **BroadcastReceiver** r_m whose $r_m.\mathcal{B}$ contains b_n **do**
        $r_m.\eta <= r_m.\eta + a_i.\eta \cdot w_k.\eta$; // inherit the access intensity // of the widget that sends b_n
      end for
    end for
  end for
end for
for each **BroadcastReceiver** r_m in entity relationship graph **do**
  for each system broadcast b_n in $r_m.\mathcal{B}$ **do**
    $r_m.\eta <= r_m.\eta + \tau \cdot$ access intensity of b_n;
  end for
end for

The whole process of computing access intensities of **BroadcastReceiver**s is described in **Algorithm 7**.

4) ESTIMATE API CALLS

For a certain API f_y, we define as $N(f_y)$ the estimated number of calls of f_y. Note that, f_y may exist in multiple execution flows starting from different entry functions of different

Algorithm 8 Process of Estimating Number of API Calls

Input: entity relationship graph
Output: estimated number of API calls
Initial: set the initial value of $N(\cdot)$ for each API as 0
for each **Activity** a_i in entity relationship graph **do**
  for each entry function e_x in $a_i.\mathcal{E}$ **do**
    for each API f_y in $e_x.\mathcal{F}$ **do**
      $N(f_y) <= N(f_y) + a_i.\eta \cdot f_y.q$;
    end for
  end for
  for each widget w_j in $a_i.\mathcal{W}$ **do**
    for each entry function e_x in $w_j.\mathcal{E}$ **do**
      for each API f_y in $e_x.\mathcal{F}$ **do**
        $N(f_y) <= N(f_y) + a_i.\eta \cdot w_j.\eta \cdot f_y.q$;
      end for
    end for
  end for
end for
for each **Service** s_k in entity relationship graph **do**
  for each entry function e_x in $s_k.\mathcal{E}$ **do**
    for each API f_y in $e_x.\mathcal{F}$ **do**
      $N(f_y) <= N(f_y) + s_k.\eta \cdot f_y.q$;
    end for
  end for
end for
for each **BroadcastReceiver** r_l in entity relationship graph **do**
  for each API f_y in $r_l.e.\mathcal{F}$ **do**
    $N(f_y) <= N(f_y) + r_l.\eta \cdot f_y.q$;
  end for
end for

entities, so $N(f_y)$ is the sum of f_y's number of estimated API calls in these execution flows.

Initially, the estimated numbers of API calls of all APIs are set as 0. Then, each **Activity**, widget, **Service** and **BroadcastReceiver** in entity relationship graph is traversed, where the estimated number of calls of each API in each entry function is accumulated. If f_y is in the execution flow of a lifecycle function of **Activity** a_i, the accumulated number of API calls is $a_i.\eta \cdot f_y.q$; if f_y is in the execution flow of an event handler function of widget w_j in **Activity** a_i, the accumulated number of API calls is $a_i.\eta \cdot w_j.\eta \cdot f_y.q$; if f_y is in the execution flow of a lifecycle function of **Service** s_k, the accumulated number of API calls is $s_k.\eta \cdot f_y.q$; if f_y is in the execution flow of the event handler function of **BroadcastReceiver** r_l, the accumulated number of API calls is $r_l.\eta \cdot f_y.q$.

The whole process of estimating number of API calls is described in **Algorithm 8**.

Specifically, **Application** of the target app is launched only once during the running period of the app. Thus, the estimated API calls in **Application** of the app can be directly obtained. For an API f_y in a lifecycle function of **Application**, its accumulated API calls is $f_y.q$, which represents the number of actual API calls of f_y in **Application**, so scaling-up or scaling-down is not required.

TABLE 5. The values of parameters in EstiDroid.

Parameter Name	Value
Ω	3
α, β, γ	0.4, 0.3, 0.3
δ	0.2
ϵ	10^{-3}
ρ	0.2
τ	0.8

Note that, the estimated number of calls for API f_y is a virtual value, which is used to compute the proportion of f_y's calls in the total number of API calls. It does not represent the actual number of calls of f_y in real environment.

We define as $\sum N(\cdot)$ the sum of estimated number of calls of each API, and define as ξ_{f_y} the proportion of f_y's calls in the total number of API calls. Then, ξ_{f_y} can be formulated as

$$\xi_{f_y} = \frac{N(f_y)}{\sum N(\cdot)} \tag{6}$$

IV. EXPERIMENTS AND EVALUATIONS

The prototype of Estidroid is implemented using Java language (J2SE 1.8), and with Apktool (v2.2.4) and Soot (v2.5) integrated. The prototype runs in a Dell PowerEdge R720 server with a 2.8 GHz Intel Xeon E5-2680 CPU and 96 GB 1866MHzRAM, which runs Ubuntu 12.04 with Linux kernel 3.8.0.

In the evaluation of the performance of EstiDroid and the accuracy of EstiDroid's estimation, we choose 300 apps from Wandoujia Android market [30], and COOLAPK [31] Android market. **(a)** Each app is firstly deployed on 10 Nexus 6 smartphones with DroidInjector [3] installed. DroidInjector is a dynamic API calls tracking tool that can track API calls during the running period of the target app without modifying Android OS. For each target app, 10 testers (the first 5 authors of this paper, and 5 invited Android users) manually operate it for a time period. The API calls generated are continuously tracked, and they are stored by DroidInjector when the running time reaches 1 hour, 6 hours, 24 hours and 48 hours, respectively; **(b)** Then, we estimate the API calls of all target apps through running EstiDroid in the server. Parallelism is not used in order to give a fair environment to evaluation the time consumption of EstiDroid. The parameters of Estidroid is illustrated in TABLE 5. For each app, beside obtaining the estimate API calls, we also measure the time consumption of static analyzer and the time consumption of estimation algorithm.

Finally, for each app, the tracked API calls and the estimated API calls are all transformed into proportions in the total number of API calls, for further evaluations.

We measure the similarity between the estimated API calls and the tracked API calls to evaluate the accuracy of EstiDroid. Cosine similarity is adopted, which is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between

TABLE 6. Partial experiment results I.

App Package Name	Time Consumption of Estimation Algorithm (ms)	Time Consumption of Static Analyzer (ms)	Cosine Similarity (vs. 1 hours Manual Testing)	Cosine Similarity (vs. 6 hours Manual Testing)	Cosine Similarity (vs. 12 hours Manual Testing)	Cosine Similarity (vs. 24 hours Manual Testing)	Cosine Similarity (vs. 48hours Manual Testing)
air.com.RustyLake.CubeEscapeHarveysBox.pj.apk	3267	76970	0.355641247	0.574294754	0.73235753	0.858807751	0.878126535
aolei.fc.apk	1852	49977	0.354222142	0.570553614	0.678719349	0.788536471	0.825692639
app.greyshirts.sslcapture.apk	3821	71941	0.34952093	0.49999805	0.675952778	0.753977951	0.796175239
cc.yidu.wallpaper.apk	4051	99789	0.425126648	0.517816735	0.770040721	0.866295812	0.891250835
cn.bevol.p.apk	1341	7863	0.32679769	0.553164871	0.636059895	0.769967241	0.797067538
cn.appoa.risenameword.apk	4487	37629	0.399963562	0.511064551	0.782150965	0.869254141	0.888807915
cn.apppark.ckj10896022.apk	1381	16221	0.375911181	0.531582635	0.733247927	0.865037965	0.884496897
cn.beekee.zhongtong.apk	4435	94463	0.421276045	0.535535179	0.752714755	0.82423574	0.872207132
cld.navi.mainframe.apk	2686	27473	0.440994267	0.514344227	0.787170295	0.843524532	0.894511698
cn.com.dreamtouch.ahc.apk	1024	97515	0.395024872	0.51417205	0.680498312	0.774056835	0.799645491
cn.com.mcdonalds.m4d.apk	1291	44906	0.434848525	0.593461306	0.782370911	0.872370298	0.891083042
cn.com.nd.s.apk	1609	38471	0.412316018	0.563292547	0.71515198	0.853767857	0.882903679
cn.com.weshare.jiekuan.apk	1565	38410	0.389284752	0.495613093	0.768903291	0.835688034	0.878746618
cn.dooone.wifihelper_cn.apk	3968	74954	0.395436586	0.45204142	0.621855922	0.750213362	0.797251182
cn.edu.zjicm.wordsnet_d.apk	1193	39976	0.388297886	0.536496865	0.676416573	0.789842607	0.827927261
cn.etouch.ecalendar.apk	1157	46794	0.396846991	0.5104778	0.734296059	0.833292597	0.86083946
cn.gamedog.terrariaassist.apk	4859	36453	0.414842838	0.509440891	0.715126566	0.833157165	0.867872047
cn.gov.jsgsj.portal.apk	2578	92907	0.404521312	0.477385399	0.654939266	0.793967295	0.837518243
cn.ibuka.manga.ui.apk	1815	40508	0.340045047	0.503711588	0.659433152	0.77781332	0.794497773
com.android.mediacenter.apk	1741	11758	0.367930841	0.564326357	0.684483952	0.785582066	0.828673065
com.autoconnectwifi.app.apk	2379	26497	0.346660859	0.490696005	0.699018023	0.782021328	0.813757885
com.autonavi.minimap.apk	733	12104	0.429600444	0.51481771	0.710729569	0.853051189	0.878528516
com.baidu.searchbox.lite.apk	1883	80032	0.358083793	0.567320404	0.73743147	0.82844089	0.850555328
com.bona.rueiguangkangtai.apk	2499	32446	0.337811234	0.48175543	0.663141758	0.808750048	0.832047375
com.buzzpia.aqua.launcher.apk	1527	63247	0.408621394	0.497490013	0.766709653	0.818985311	0.871260969
com.cameras.dreamycam.apk	2964	64687	0.348063472	0.480144838	0.677851536	0.814086403	0.830700411
com.changba.apk	3053	37478	0.40222985	0.465129991	0.715075288	0.795355731	0.827633435
com.coohuaclient.apk	3518	75169	0.335552749	0.567858498	0.669440224	0.784344143	0.832637095
com.dada.mobile.shop.android.apk	1680	38726	0.404008328	0.518130268	0.673902551	0.80968267	0.833006862
com.dalongtech.cloud.apk	2025	21619	0.391607092	0.461854137	0.641912656	0.758991065	0.807437303
com.dmzj.manhua.apk	2010	44684	0.329534714	0.504958666	0.694318166	0.797605166	0.819738095
com.doc360.client.apk	1576	18173	0.31481086	0.433355598	0.685361299	0.751306716	0.785064489
com.dushe.movie.apk	6598	81618	0.381101025	0.56438478	0.660467453	0.783194902	0.807417425
com.fangku.feiqubuke.apk	1902	56181	0.375583266	0.467446706	0.655238339	0.784497515	0.812950793
com.gotokeep.keep.apk	1360	15992	0.423310875	0.547663768	0.684792643	0.828735375	0.851732143
com.gvsoft.gofun.apk	2059	22448	0.403963776	0.628887927	0.748547576	0.870906314	0.899696606
com.hipu.yidian.apk	1168	12040	0.352113393	0.579256085	0.722803511	0.81062076	0.844396625
com.hsrjyhq.app.apk	1617	29828	0.388091213	0.52819053	0.697275913	0.756858381	0.805168491
com.huahan.publicmove.apk	2070	59642	0.395284184	0.587203899	0.686685219	0.838988479	0.880365665
com.huawei.gallery.apk	1129	23601	0.327623775	0.48939312	0.665051753	0.768812898	0.817016895
com.idea.android.eye.protector.apk	1427	14771	0.442002772	0.563111532	0.701016397	0.833617229	0.884005545
Average	2324	45755	0.382548987	0.523605459	0.702406371	0.810201013	0.843424689
Min	733	7863	0.31481086	0.433355598	0.621855922	0.750213362	0.785064489
Max	6598	99789	0.442002772	0.628887927	0.787170295	0.872370298	0.899696606
Stand Deviation	1267.246	26716.386	0.034972493	0.043401188	0.043172532	0.037020767	0.034599772

them [32]. In our scenario, each API represents a dimension of the multi-dimensional space, and the proportion of the API's calls is the value on the dimension. Thus, the estimated API calls and the tracked API calls are two vectors in the space. Because the proportions of API calls are all positive values, the value range of cosine similarity is [0, 1]. If the estimated API calls is very close to the tracked API calls, the angle between the two vectors is very small, so the similarity tends to 1, else 0.

The experiment results for the 300 apps show that vs. 48 hours manual testing, EstiDroid reaches 84.06% average similarity between estimated and tracked API calls, and the minimum and maximum similarities are 77.02% and 90.74%, and standard deviation 0.034727691, respectively; vs. 1 hour manual testing, the apps have 37.96% average similarity, 28.25% minimum similarity, 46.42% maximum similarity, and 0.030762336 standard deviation; vs. 6 hours manual testing, the results are 52.86%, 43.57%, 64.85% and 0.046674688; vs. 12 hours manual testing, the results are 69.98%, 61.52%, 80.84% and 0.038804211; vs. 24 hours manual testing, the results are 80.94%, 74.01%, 89.05% and 0.035394057.

The standard deviation of the similarities for the 300 apps indicates the performance of EstiDroid is quite stable. The average time consumptions of the estimation algorithm and

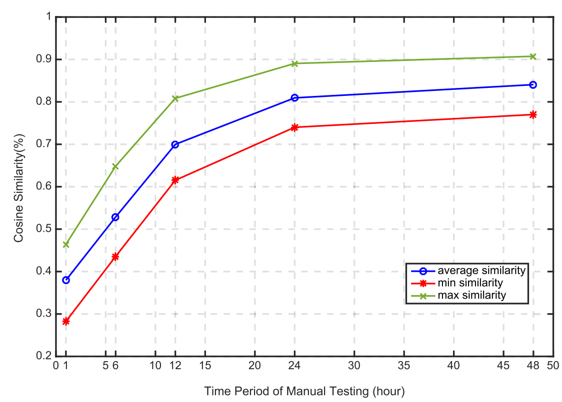


FIGURE 6. The cosine similarity as the increase of time period of manual testing.

the static analyzer are 2584ms and 46658ms, respectively. Thus. the total average time consumption of EstiDroid is 2584ms + 46658ms = 49242ms. The experiment results demonstrate that EstiDroid can largely reduce the time needed by manual API calls tracking, while keeping a high and stable similarity with the testing results in actual environments.

It can be found that the similarity rises as the time period of manual testing increases, as shown in Fig. 6. This is

TABLE 7. Partial experiment results II.

App Package Name	Time Consumption of Estimation Algorithm (ms)	Time Consumption of Static Analyzer (ms)	Cosine Similarity člvs. 1 hours Manual Testing)	Cosine Similarity člvs. 6 hours Manual Testing)	Cosine Similarity člvs. 12 hours Manual Testing)	Cosine Similarity člvs. 24 hours Manual Testing)	Cosine Similarity člvs. 48hours Manual Testing)
com.iflytek.vflynote.apk	1727	53004	0.370988755	0.545804351	0.632816639	0.758588946	0.791020799
com.itings.myradio.apk	3970	95067	0.367474813	0.537827375	0.66031898	0.777943368	0.811202678
com.jifen.qukan.apkąć	2077	83271	0.341862622	0.530389804	0.721430681	0.80438264	0.837898584
com.jingdian.tianxiameishi.android.apk	3618	95522	0.366226686	0.477986004	0.705803074	0.834756132	0.859687057
com.julanling.workschedule.apk	3934	93757	0.40440912	0.573738551	0.730458343	0.870065055	0.900688462
com.kmxs.reader.apk	7139	90169	0.335672672	0.453823193	0.676040067	0.737071462	0.782453781
com.kuaibao.kuaidi.apk	1344	10884	0.403030391	0.490150092	0.64322022	0.775935278	0.81420281
com.lemon.faceu.apk	1055	28792	0.35314784	0.528048074	0.704621994	0.8150853	0.836843224
com.lenovo.leos.cloud.sync.apk	2800	51793	0.347792855	0.519205048	0.713803431	0.805720114	0.828078227
com.longi.hrssc.apk	2451	76624	0.342907876	0.499007729	0.669608662	0.812060442	0.853004666
com.mddjob.android.apk	1274	9977	0.416288111	0.552939209	0.732124091	0.85158068	0.904974155
com.mfw.roadbook.apk	1410	11566	0.354389056	0.530415268	0.644131316	0.735259931	0.778877045
com.mianfeizs.book.apk	6289	16549	0.409702783	0.575384788	0.733863227	0.847319382	0.900445677
com.miantan.myoface.apk	2416	22367	0.404293018	0.510685918	0.741345724	0.827311187	0.851143196
com.moji.mjweather.light.apk	1606	67548	0.349932317	0.559725863	0.678304823	0.807663689	0.8292235
com.pplive.androidphone.sport.apk	6026	37918	0.369423516	0.594096952	0.731742733	0.847187582	0.888037298
com.qihoo.freewifi.apk	1713	19775	0.355695821	0.592530694	0.768161049	0.854202183	0.887021997
com.qq.ac.android.apk	2156	94740	0.38119572	0.553209207	0.755476324	0.847101599	0.864389387
com.resou.reader.apk	1003	30805	0.385198949	0.47371275	0.720403992	0.773676187	0.819572232
com.roamingsoft.manager.apk	5201	80670	0.342389463	0.514781361	0.698346807	0.774965428	0.798110637
com.sankuai.mhotel.apk	9788	97624	0.344678098	0.473628257	0.7120644	0.779378005	0.81100729
com.shoujiduoduo.ringtone.apk	6765	68021	0.383369159	0.45041932	0.65235863	0.74386238	0.788825429
com.sing.client.apk	2948	24657	0.354345159	0.505579321	0.716603734	0.858165944	0.879268385
com.sinyee.babybus.recommendapp.apk	1277	24334	0.338254398	0.52580139	0.710836593	0.818006303	0.83726336
com.sirma.mobile.bible.android.apk	5125	46795	0.345224403	0.588509094	0.713577985	0.818944242	0.856636236
com.snda.wifilocating.apk	1238	29785	0.375644821	0.505268738	0.764516573	0.846523541	0.881795355
com.songheng.eastnews.apk	1482	58226	0.33571266	0.537140255	0.707897806	0.79078982	0.828920147
com.spider.film.apk	2710	87912	0.368036486	0.601531331	0.714372707	0.821138008	0.868010579
com.ss.android.article.lite.apk	5843	56052	0.355191286	0.445385927	0.641738863	0.763062628	0.798182666
com.syezon.wifi.apk	1778	19008	0.401097384	0.547620714	0.685958383	0.779274917	0.818566089
com.tencent.gallerymanager.apk	3482	73411	0.431193243	0.555376897	0.688184416	0.832202959	0.862386486
com.tencent.radio.apk	1243	13142	0.350299455	0.590159673	0.739425204	0.823117439	0.862806539
com.tencent.wifimanager.apk	2483	65361	0.366882625	0.553599676	0.668250496	0.798461071	0.818934432
com.thunder.ktvdaren.apk	1316	16875	0.397695707	0.529442639	0.700468159	0.773297208	0.818303924
com.tplink.cloudrouter.apk	1497	19243	0.359574531	0.582579889	0.696675653	0.846210254	0.864361853
com.tujia.hotel.apk	1275	26011	0.399413616	0.472034274	0.679406596	0.7633238	0.806896195
com.UCMobile.apk	4405	61332	0.43143346	0.512819488	0.683467611	0.824361702	0.87511858
com.uelive.showvideo.activity.apk	1409	11508	0.331725165	0.485711167	0.655259585	0.776482608	0.819074481
com.uxin.usedcar.apk	1753	12021	0.402169099	0.609971395	0.70133275	0.863454369	0.895699552
com.vv51.mvbox.apk	1919	30181	0.363113073	0.516610872	0.713847291	0.783994135	0.825256984
com.wuba.apk	1465	21550	0.347269468	0.474338524	0.643237765	0.75136485	0.789248792
Average	2937	47167	0.370349895	0.528707099	0.698817156	0.805202263	0.840083872
Min	1003	9977	0.331725165	0.445385927	0.632816639	0.735259931	0.778877045
Max	9788	97624	0.43143346	0.609971395	0.768161049	0.870065055	0.904974155
Stand Deviation	2070.745	30241.068	0.027587762	0.044386098	0.034949541	0.037572044	0.036082633

because the statistical characteristics of the running behavior of an app is built based on the long running period of the app. Too short operation period can not reflex adequately the app's running behavior. Some features of the app may not be activated during the short period, so the cosine similarities vs. 1 hour and 6 hours manual testing are low. As the manual testing period increases, more and more potential API calls are triggered. The statistical characteristics of the running behavior of the app are expressed more and more complete. The increase of the cosine similarity slows down as the operation period increases, especially from 24 hours to 48 hours. It is because the API calls have been already triggered sufficiently and become stable.

Due to the page limitation of our paper, we gives the detailed results of 82 apps from the 300 apps. The results are shown in TABLE 6 and TABLE 7.

V. CONCLUSION

In this paper, we proposed EstiDroid, a static analysis method which estimates API calls of Android apps by statically analyzing the apps without actually running them. EstiDroid contains a static analyzer and an estimation algorithm. It's an approach to high-speed API calls tracking through estimation based on static analysis.

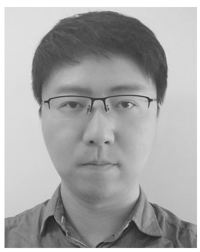
When analyzing a target app, the static analyzer uses Apktool and Soot to extract the .APK file and convert the file into Jimple files. Serval types of static information are output by the XML parser and the Java decompiler of the static analyzer. Then, according to the static information, the estimation algorithm first constructs entity description models for all entities, then composes the entity relationship graph of the app. The access intensities of widgets, **Activity**s, **Services**, and **BroadcastReceiver**s are computed using different mechanisms including a PageRank-based algorithm. Finally, the estimated number of API calls are counted based on the access intensities and the frequency of each API.

In experiments, we tested 300 apps picked from Android App Markets. The API calls are collected by manually operating each app on smartphones for 1 hour, 6 hours, 12 hours, 24 hours and 48 hours, and are estimated by EstiDroid for each app. Multiple metrics are evaluated to prove the high performance of EstiDroid. The experiment results show EstiDroid only consumed 49242ms on average, and reached 84.06% average and 90.74% maximum similarity with tracked API calls of apps running in real environment (vs. 48 hours manual testing).

Our future work aims at increasing the accuracy of EstiDroid's estimation and decreasing EstiDroid's time consumption. We plan to further improve EstiDroid using some new technologies. If possible, we would like to apply EstiDroid to malware detection systems, user behavior mining systems, and energy consumption prediction systems, in order to boost the efficiency of these systems.

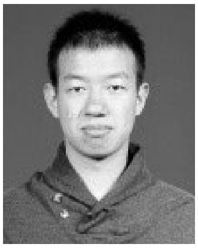
REFERENCES

[1] (2019). *Smartphone OS Market Share*. [Online]. Available: http://www.idc.com/promo/smartphone-market-share/os
[2] Statista. (2017). *Number of Available Applications in the Google Play Store From December 2009 to July 2019*. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store
[3] W. Fan, Y. Sang, D. Zhang, R. Sun, and Y. Liu, ''DroidInjector: A process injection-based dynamic tracking system for runtime behaviors of Android applications,'' *Comput. Secur.*, vol. 70, pp. 224–237, Sep. 2017.
[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, ''Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps,'' *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, ''Analyzing inter-application communication in Android,'' in *Proc. 9th Int. Conf. Mobile Syst., Appl., Services (MobiSys)*. New York, NY, USA: ACM, 2011, pp. 239–252.
[6] F. Wei, S. Roy, X. Ou, and Robby, ''Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: ACM, 2014, pp. 1329–1341.
[7] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer, ''R-Droid: Leveraging Android app analysis with static slice optimization,'' in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur. (ASIA CCS)*. New York, NY, USA: ACM, 2016, pp. 129–140.
[8] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, ''IccTA: Detecting inter-component privacy leaks in Android apps,'' in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1. Piscataway, NJ, USA: IEEE Press, May 2015, pp. 280–291.
[9] 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. Test. Anal. (ISSTA)*. New York, NY, USA: ACM, 2016, pp. 318–329.
[10] S. Calzavara, I. Grishchenko, and M. Maffei, ''HornDroid: Practical and sound static analysis of Android applications by SMT solving,'' in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS P)*, Mar. 2016, pp. 47–62.
[11] Z. Yang and M. Yang, ''LeakMiner: Detect information leakage on Android with static taint analysis,'' in *Proc. 3rd World Congr. Softw. Eng.*, Nov. 2012, pp. 101–104.
[12] Z. Zhao and F. C. Colon Osono, '''TrustDroid': Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking,'' in *Proc. 7th Int. Conf. Malicious Unwanted Softw.*, Oct. 2012, pp. 135–143.
[13] BlackHat. *Reverse Engineering With Androguard*. Accessed: Mar. 29, 2013. [Online]. Available: https://code.google.com/androguard
[14] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, ''Detecting repackaged smartphone applications in third-party Android marketplaces,'' in *Proc. 2nd ACM Conf. Data Appl. Secur. Privacy (CODASKY)*. New York, NY, USA: ACM, 2012, pp. 317–326.
[15] Google. (2017). *Android Bouncer*. [Online]. Available: https://en.wikipedia.org/wiki/google_play#application_security
[16] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, ''TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones,'' *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, Jun. 2014.
[17] M. Dam, G. Le Guernic, and A. Lundblad, ''TreeDroid: A tree automaton based approach to enforcing data processing policies,'' in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: ACM, 2012, pp. 894–905.
[18] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, ''SmartDroid: An automatic system for revealing UI-based trigger conditions in Android applications,'' in *Proc. 2nd ACM Workshop Secur. Privacy Smartphones Mobile Devices (SPSM)*. New York, NY, USA: ACM, 2012, pp. 93–104.
[19] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen, '''Little brothers watching you:' Raising awareness of data leaks on smartphones,'' in *Proc. 9th Symp. Usable Privacy Secur. (SOUPS)*. New York, NY, USA: ACM, 2013, pp. 12:1–12:11.
[20] V. Rastogi, Y. Chen, and W. Enck, ''AppsPlayground: Automatic security analysis of smartphone applications,'' in *Proc. 3rd ACM Conf. Data Appl. Secur. Privacy (CODASPY)*. New York, NY, USA: ACM, 2013, pp. 209–220.
[21] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan, ''On tracking information flows through JNI in Android applications,'' in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 180–191.
[22] D. Schreckling, J. Köstler, and M. Schaff, ''Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for Android,'' *Inf. Secur. Tech. Rep.*, vol. 17, no. 3, pp. 71–80, 2013.
[23] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, '''Andromaly': A behavioral malware detection framework for Android devices,'' *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, Feb. 2012.
[24] M. Zheng, M. Sun, and J. C. S. Lui, ''DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability,'' in *Proc. Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, Aug. 2014, pp. 128–133.
[25] R Team. (2017). *Official Site of Robotium*. [Online]. Available: https://robotium.com
[26] Google. (2017). *Official Site of Monkeyrunner in Android Studio*. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner .
[27] R. Wiśniewski and C. Tumbleson. (2017). *Official Site of Apktool*. [Online]. Available: https://ibotpeaches.github.io/apktool/
[28] SableResearchGroup. (2017). *Official Site of Soot*. [Online]. Available: https://sable.github.io/soot/
[29] L. Page, S. Brin, R. Motwani, and T. Winograd, ''The pagerank citation ranking: Bringing order to the Web,'' Stanford InfoLab, Stanford, CA, USA, Tech. Rep., 1999.
[30] *Wandoujia Android Market*. Accessed: 2020. [Online]. Available: https://www.wandoujia.com/
[31] *Coolapk Android Market*. Accessed: 2020. [Online]. Available: https://www.coolapk.com/
[32] (2017). *Cosine Similarity*. Accessed: 2020. [Online]. Available: https://en.wikipedia.org/wiki/cosine_similarity



WENHAO FAN (Member, IEEE) received the B.E. and Ph.D. degrees from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2008 and 2013, respectively. He is currently an Associate Professor with the School of Electronics Engineering, BUPT. His main research topics include mobile security, mobile cloud/edge computing, parallel computing/ transmission, and the software engineering for mobile Internet.

DAISHUAI ZHANG received the B.E. and M.Eng. degrees from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2015 and 2018, respectively. His main research topics include mobile security and the software engineering for mobile Internet.



YE CHEN received the B.E. degree from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2017, where he is currently pursuing the M.S. degree with the School of Electronics Engineering. His main research topics include mobile security and the software engineering for mobile Internet.

FAN WU (Member, IEEE) received the B.E. degree from the University of Electronic Science and Technology of China, Chengdu, China, in 2004, and the Ph.D. degree from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2009. She is currently an Associate Professor with the School of Electronics Engineering, BUPT. Her main research topics include network and information security, and wireless sensor networks.

YUAN'AN LIU (Member, IEEE) received the B.E., M.Eng., and Ph.D. degrees in electrical engineering from the University of Electronic Science and Technology of China, Chengdu, China, in 1984, 1989, and 1992, respectively. He is currently a Professor with the Beijing University of Posts and Telecommunications (BUPT), where he is the Dean of the School of Electronics Engineering. His main research topics include mobile security, pervasive computing, wireless communications, and electromagnetic compatibility. He is a Fellow of the Institution of Engineering and Technology, U.K., the Vice Chairman of the Electromagnetic Environment and Safety of the China Communication Standards Association, the Vice Director of the Wireless and Mobile Communication Committee, Communication Institute of China, and a Senior Member of the Electronic Institute of China.