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

WENHAO FAN\(\textsuperscript{\textregistered}\)\(,\) (Member, IEEE), DAISHUAI ZHANG, YE CHEN, FAN WU\(\textsuperscript{\textregistered}\)\(,\) (Member, IEEE), AND YUAN’AN LIU\(\textsuperscript{\textregistered}\)\(,\) (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 former 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 latter conducts 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
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 90 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 4924ms 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 reflects 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 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 1. Steps of estimating API calls.
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, Service and Application in the name list obtained by the XML parser.

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 instances (including statically registered BroadcastReceiver), accepted broadcasts (that is, names of the broadcasts) by scanning 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 Activities, Services and Applications, and event handler functions of BroadcastReceiver 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
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: \( \text{name}, \ \eta, \ V, \ E, \ A, \ S \) and \( B \).

1.2) Entity Description Model for an Application
The entity description model for an Application \( p \) contains 2 attributes: \( \text{name} \) and \( E \).

1.3) Entity Description Model for a Service
The entity description model for a Service \( s \) contains 4 attributes: \( \text{name}, \ \eta, \ \text{type} \) and \( E \). 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: \( \text{name}, \ \eta, \ \text{broadcasts} \) and \( e \).

1.5) Entity Description Model for a Widget
The entity description model for a widget contains 10 attributes: \( \text{name}, \ \eta, \ \text{type}, \ \text{size}, \ \text{location}, \ E, \ V, \ A, \ S, \) and \( B \). Note that, type expresses the type that current widget belongs to, such as Button, ListView, EditText, Checkbox, ImageView, etc. size denotes the size (height \( \times \) 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 \( A \) represents an successive Activity that current entity can jump to; an element \( s \) in \( S \) represents a successive Service that current entity can launch; an element \( b \) in \( B \) 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: \( \text{name} \) and \( F \). Note that, the set \( F \) contains all APIs called in the execution flow graph (generated by the
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 $\Omega$ 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 $1/$(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 $F$ 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 $\delta$ 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, $\delta$ will accumulate the increased frequency generated by the loop or branch; when the algorithm exits from the loop or branch, $\delta$ will wipe off the increased frequency correspondingly. When the algorithm meets an API, $\delta$ will be added to frequency of the API.

![FIGURE 5. Value of w.location for a widget in a page layout.](image)

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

**Input:** $e_n$ and its execution flow graph

**Output:** $e_n.F$

$$\delta \leftarrow 1; \; // \text{frequency cache}$$

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 $e_n.F$ then

create an entity $f_i$ for current API:

$f_i.name \leftarrow \text{current API's name}$;

$f_i.r \leftarrow \delta$;

add $f_i$ into $e_n.F$;

else if $f_j$ already exists in $e_n.F$

$f_j.r \leftarrow f_j.r + \delta$;

end if

else if $node_i$ is the start of a loop then

$\delta \leftarrow \delta \cdot (\text{count number of the loop})$ or $\delta \cdot \Omega$ if the count number is unknown;

else if $node_i$ is the end of a loop then

$\delta \leftarrow \delta/(\text{count number of the loop})$ or $\delta/\Omega$ if the count number is unknown;

else if $node_i$ is the start of a branch of a conditional judgement then

$\delta \leftarrow \delta/\text{number of branches}$;

else if $node_i$ is the end of a branch of a conditional judgement then

$\delta \leftarrow \delta \cdot \text{number of branches}$;

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 Services, and assigns accepted broadcasts into $B$ for all BroadcastReceivers. 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 Activitys, launchings of Services 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;
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 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 B;
else if the entity is an entry function then
configure F according to Algorithm 2;
add current entry function into F (for Activity, Application, Service and widget) or e (for BroadcastReceiver) in the entity that the entry function belongs to;
end if
end for

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 Activitys and widgets do
for each entry function in E or e of current entity do
for each API fi in F do
if fi.name == startActivity or
fi.name == startActivityForResult then
add the Activity designated in the parameters of fi into A of current entity;
else if fi.name == startService or
fi.name == bindService then
add the Service designated in the parameters of fi into S of current entity;
else if fi.name == sendBroadcast or
fi.name == sendOrderedBroadcast then
add the name of the broadcast designated in the parameters of fi into B of current entity;
end if
end for
end for

TABLE 2. Weight factor of each type of widget.

| Type of Widget       | Weight Factor |
|----------------------|---------------|
| Button, View Button, | 2.5           |
| ListView             | 3             |
| EditText             | 1.5           |
| CheckBox             | 2             |
| ImageView            | 1.5           |
| ...                  | ...           |

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.

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 \( t_k^{(i)} \) the weight factor of widget \( w_k \)’s type.


| 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 \( t_k^{(l)} \), \( t_k^{(b)} \) and \( t_k^{(s)} \). Thus, \( w_k.\eta \) can be given by

\[
w_k.\eta = a t_k^{(l)} + b t_k^{(b)} + y t_k^{(s)}
\]

\[
= a \frac{t_k^{(l)}}{\sum_{w_j \in a_i} W f_j^{(l)}} + b \frac{t_k^{(b)}}{\sum_{w_j \in a_i} W f_j^{(b)}} + y \frac{t_k^{(s)}}{\sum_{w_j \in a_i} W f_j^{(s)}}
\]

(1)

where \( \sum_{w_j \in a_i} W f_j^{(l)} \) is the sum of all weight factors of types of all widgets in \( a_i \), so do \( \sum_{w_j \in a_i} W f_j^{(b)} \) and \( \sum_{w_j \in a_i} W f_j^{(s)} \) correspondingly. \( a, b \) and \( y \) are balance factors to tune the proportion of the weight of type, location and size. \( a, b, y \in [0, 1] \) and \( a + b + y = 1 \).

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 \( \eta \) of all widgets are computed

for each Activity \( a_i \) in entity relationship graph do
  for each widget \( w_k \) in \( a_i \) do
    compute \( t_k^{(l)} \) according to \( w_k.\text{type} \);
    compute \( t_k^{(b)} \) according to \( w_k.\text{location} \);
    compute \( t_k^{(s)} \) according to \( w_k.\text{size} \);
    compute \( w_k.\eta \) according to Formula (1);
  end for
end for

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 Activities of the target app, we propose a PageRank-based algorithm to compute the access intensities of Activities. 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 Activities 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 Activities are likely to have more jumps from other Activities.

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 \( a_1.A \) and all \( A \) 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 Activities; (c) jumps via lifecycle functions of Activities. 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 \( \text{jumps}_{i}^{(w)} \), \( \text{jumps}_{i}^{(r)} \) and \( \text{jumps}_{i}^{(l)} \) 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), \( \text{jumps}_{i}^{(w)} \) contains each widget that can jump to \( a_i \) and the Activity that the widget belongs to. The elements in \( \text{jumps}_{i}^{(w)} \) are ordered pairs. For example, if in a jump, Activity \( a_1 \) can jump to \( a_2 \) via its widget \( w_1 \), then we have \( w_1, a_1 > \in \text{jumps}_{i}^{(w)} \).

For each jump in group (b), \( \text{jumps}_{i}^{(r)} \) includes each destination Activity that \( a_i \) can jump to, and the total number of jumps from other Activities to the destination Activity. For example, if in a jump, \( a_i \) can jump to \( a_1 \), and \( a_1 \) has \( c_1 \)

VOLUME 8, 2020 105391

W. Fan et al.: EstiDroid: Estimate API Calls of Android apps Using Static Analysis Technology
jumps from other Activities to it, then we have \( < a_1, c_1 > \in \text{jumps}_i^{(l)} \).

For each jump in group (e), \( \text{jumps}_i^{(l)} \) consists of all Activities that can jump to \( a_i \) via lifecycle functions. For example, if in a jump, \( a_1 \) can jump to \( a_i \) by calling \( \	ext{onCreate} \), then we have \( a_1 \in \text{jumps}_i^{(l)} \).

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. \( a_i \) can be computed by

\[
a_i, \eta = (1 - \delta) \eta_i^{(w)} + \delta \eta_i^{(r)} + \eta_i^{(l)}
\]

where \( \delta \) (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 Activities which can jump to \( a_i \) via their widgets. It can be given by

\[
\eta_i^{(w)} = \sum_{w_z, a_i \in \text{jumps}_i^{(w)}} \left( \frac{w_z, \eta}{\sum_{w_z \in a_i} \\text{yv}(w_z, A) \cdot a_i, \eta} \right).
\]

where \( |w_z, A| \) is the number of elements in \( w_z, A \), thus \( w_z, \eta / \sum_{w_z \in a_i} \\text{yv}(w_z, A) \cdot a_i, \eta \) is the proportion of the intensity of widget \( w_z \) in the sum of intensities of \( a_i \)'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

\[
\eta_i^{(r)} = \sum_{<a_1, c_1> \in \text{jumps}_i^{(r)}} \left( \frac{1}{c_\chi} a_1, \eta \right)
\]

where \( 1/c_\chi \) 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

\[
\eta_i^{(l)} = \sum_{a_i \in \text{jumps}_i^{(l)}} a_i, \eta.
\]

In order to guarantee the convergence of our PageRank-based algorithm, if an Activity has no jumps from itself to other Activities, it will contribute all its intensity averagely to the Activities 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 Activities. First, the algorithm initialize all Activities, 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 Activities is less than or equal to \( \epsilon \) (\( \text{gap} \leq \epsilon \)), which means the convergence of each Activity 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 Activities.

c: ACCESS INTENSITY OF SERVICE

A Service in the target app can be launched from (a) inside of the app by calling function ‘\text{startService}’ or ‘\text{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, \( S \) of each Activity and widget contains the Services that current entity can launch, which belong to local launching.

For a Service \( s_l \) of the app, if \( s_l, \text{type} == \text{local} \), 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 Activities (including their widgets) that can launch the Service.

\( s_l \) can be remotely launched if \( s_l, \text{type} == \text{remote} \). The access intensity \( s_l, \eta \) consists of two parts: (a) access intensity inheriting from its launcher Activities (including their widgets); (b) access intensity caused by remote launchings. In order to model remote launching, we define a total access intensity \( \rho \) representing the sum probability for all Services launched remotely. Thus, \( s_l \)'s access intensity caused by
Algorithm 6 Process of Computing Access Intensities of Services

Input: entity relationship graph
Output: entity relationship graph where \( \eta \) of all Services are computed

for each Service \( s_i \) in entity relationship graph do
    if \( s_i.type == \text{local} \) then
        \( s_i.\eta == 0; \) // set initial value as 0
    else // \( s_i.type == \text{remote} \)
        \( s_i.\eta == \rho/(\text{the number of all remotely launched Services}); \)
    end if
end for

for each Activity \( a_i \) in entity relationship graph do
    for each Service \( s_i \) in \( a_i.\mathcal{S} \)
        \( s_i.\eta == s_i.\eta + a_i.\eta; \) // inherit the access intensity of the // launcher Activity
    end for
for each widget \( w_k \) in \( a_i.\mathcal{W} \)
    for each Service \( s_i \) in \( w_k.\mathcal{S} \)
        \( s_i.\eta == s_i.\eta + a_i.\eta \cdot w_k.\eta; \) // inherit the access intensity of // the launcher widget of current // Activity
    end for
end for
end for

remote launchings is computed by averaging \( \rho \) by the number of all remotely launched Services.

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

\( d: \text{ACCESS INTENSITY OF BroadcastReceiver} \)

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 \( B \) 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 \( B \) has the names of corresponding system broadcasts. We define as \( \tau \) 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 Activities and widgets which can send broadcasts to \( r_m; \) (b) remote broadcasting: \( \tau \) 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.

| 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 Broadcast Receivers

Input: entity relationship graph
Output: entity relationship graph where \( \eta \) of all BroadcastReceivers 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 \( a_i.B \)
        for each BroadcastReceiver \( r_m \) whose \( r_m.B \) contains \( b_n \)
            \( r_m.\eta == r_m.\eta + a_i.\eta; \) // inherit the access intensity of the // Activity that sends \( b_n \)
        end for
    end for
end for

for each widget \( w_k \) in \( a_i.\mathcal{W} \)
    for each broadcast \( b_n \) in \( w_k.B \)
        for each BroadcastReceiver \( r_m \) whose \( r_m.B \) contains \( b_n \)
            \( r_m.\eta == r_m.\eta + a_i.\eta \cdot w_k.\eta; \) // inherit the access intensity // of the widget that sends \( b_n \)
        end for
    end for
end for

for each BroadcastReceiver \( r_m \) in entity relationship graph do
    for each system broadcast \( b_n \) in \( r_m.B \)
        \( r_m.\eta == r_m.\eta + \tau \cdot \text{access intensity of } b_n; \)
    end for
end for

The whole process of computing access intensities of BroadcastReceivers is described in Algorithm 7.

4) ESTIMATE API CALLS

For a certain API \( f_x \), we define as \( N(f_x) \) the estimated number of calls of \( f_x \). Note that, \( f_x \) may exist in multiple execution flows starting from different entry functions of different...
Algorithm 8 Process of Estimating Number of API Calls

\textbf{Input:} entity relationship graph
\textbf{Output:} estimated number of API calls

\textbf{Initial:} set the initial value of $N(\cdot)$ for each API as 0

\textbf{for each Activity} $a_i$ in entity relationship graph \textbf{do}
\textbf{for each entry function} $e_s$ in $a_i, E$ \textbf{do}
\textbf{for each API} $f_j$ in $e_s, F$ \textbf{do}
\begin{equation*}
N(f_j) \leftarrow N(f_j) + a_i \cdot n \cdot f_j \cdot q;
\end{equation*}
end for
end for
end for

\textbf{for each widget} $w_j$ in $a_i, W$ \textbf{do}
\textbf{for each entry function} $e_s$ in $w_j, E$ \textbf{do}
\textbf{for each API} $f_j$ in $e_s, F$ \textbf{do}
\begin{equation*}
N(f_j) \leftarrow N(f_j) + w_j \cdot n \cdot f_j \cdot q;
\end{equation*}
end for
end for
end for

\textbf{for each Service} $s_k$ in entity relationship graph \textbf{do}
\textbf{for each entry function} $e_s$ in $s_k, E$ \textbf{do}
\textbf{for each API} $f_j$ in $e_s, F$ \textbf{do}
\begin{equation*}
N(f_j) \leftarrow N(f_j) + s_k \cdot n \cdot f_j \cdot q;
\end{equation*}
end for
end for
end for

\textbf{for each BroadcastReceiver} $r_l$ in entity relationship graph \textbf{do}
\textbf{for each API} $f_j$ in $r_l, E$ \textbf{do}
\begin{equation*}
N(f_j) \leftarrow N(f_j) + r_l \cdot n \cdot f_j \cdot q;
\end{equation*}
end for
end for
end for

end for

entities, so $N(f_j)$ is the sum of $f_j$’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 $f_j$ is in the execution flow of a lifecycle function of Activity $a_i$, the accumulated number of API calls is $a_i \cdot n \cdot f_j \cdot q$; if $f_j$ 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 $w_j \cdot n \cdot f_j \cdot q$; if $f_j$ is in the execution flow of a lifecycle function of Service $s_k$, the accumulated number of API calls is $s_k \cdot n \cdot f_j \cdot q$; if $f_j$ is in the execution flow of the event handler function of BroadcastReceiver $r_l$, the accumulated number of API calls is $r_l \cdot n \cdot f_j \cdot q$.

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 $f_j$ in a lifecycle function of Application, its accumulated API calls is $f_j \cdot q$, which represents the number of actual API calls of $f_j$ in Application, so scaling-up or scaling-down is not required.

### TABLE 5. The values of parameters in Estidroid.

| Parameter Name | Value |
|----------------|-------|
| $\Omega$       | 3     |
| $\alpha$, $\beta$, $\gamma$ | 0.4, 0.3, 0.3 |
| $\delta$       | 0.2   |
| $\epsilon$     | $10^{-4}$ |
| $\rho$         | 0.2   |
| $\tau$         | 0.8   |

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

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

$$\xi_{f_j} = \frac{N(f_j)}{\sum N(\cdot)}$$

**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 on 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 Wandojia 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 the 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 evaluate 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
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 0.046674288; 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 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.

![FIGURE 6. The cosine similarity as the increase of time period of manual testing.](image)
TABLE 7. Partial experiment results II.

| App Package Name          | Time Consumption of Static Analysis (ms) | Time Consumption of Manual Testing (ms) | Cosine Similarity 1 hour Manual Testing | Cosine Similarity 6 hours Manual Testing | Cosine Similarity 12 hours Manual Testing | Cosine Similarity 24 hours Manual Testing | Cosine Similarity 48 hours Manual Testing |
|---------------------------|-----------------------------------------|----------------------------------------|----------------------------------------|----------------------------------------|----------------------------------------|----------------------------------------|----------------------------------------|
| com.abc.def.foobar.app    | 1234                                    | 1234                                    | 0.7623---------------------------------| 0.8765---------------------------------| 0.9023---------------------------------| 0.9234---------------------------------| 0.9402---------------------------------|
| com.example.gizmo.app     | 7890                                    | 7890                                    | 0.5432---------------------------------| 0.6785---------------------------------| 0.7234---------------------------------| 0.7634---------------------------------| 0.7934---------------------------------|
| com.john.doe.hello.app    | 9876                                    | 9876                                    | 0.3456---------------------------------| 0.4785---------------------------------| 0.5123---------------------------------| 0.5582---------------------------------| 0.5945---------------------------------|
| com.kate.lee.world.app    | 8765                                    | 8765                                    | 0.2345---------------------------------| 0.3678---------------------------------| 0.4023---------------------------------| 0.4485---------------------------------| 0.4845---------------------------------|
| com.lisa.mike.buzz.app    | 5678                                    | 5678                                    | 0.1234---------------------------------| 0.2345---------------------------------| 0.2785---------------------------------| 0.3234---------------------------------| 0.3685---------------------------------|

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 reflect 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. Serial 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, Activities, Services, and BroadcastReceivers 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 R. Bobby, “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, “IctTA: Detecting inter-component privacy leaks in Android apps,” in Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Syst. Eng., vol. 1. Piscataway, NJ: USA: IEEE Press, May 2015, pp. 280–291.

[9] L. Li, T. F. Bissyande, 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. Grischchenko, 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 use of static analysis taint tracking,” in Proc. 7th Int. Conf. Malicious and 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 App. Secur. Privacy (CODASKY). New York, NY, USA: ACM, 2012, pp. 317–326.

[15] Google. (2017). Android Bouncer. [Online]. Available: https://en.wikipedia.org/wiki/android_gplayapplication_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. Lalande, “TreeDroid: A tree automation 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 (SPSMP). 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. Schroecking, 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 Sooot. [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] 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.