Contextual Approach for Identifying Malicious Inter-Component Privacy Leaks in Android Apps

Daojuan Zhang*†, Yuanfang Guo*‡, Dianjie Guo*†, Rui Wang*, Guangming Yu§
*State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, 100093, China
†School of Cyber Security, University of Chinese Academy of Sciences, Beijing, 100049, China
§Chinese research institute of general technology, Beijing, 100055, China
Email: {zhangdaojuan, guoyuanfang}@iie.ac.cn

***Abstract*—Inter-Component Communication (ICC) enables developers to create rich and innovative applications in Android platform. However, some privacy problems occur because of the interactions among multiple components. Since the flow of sensitive data across components may be legal or malicious, it is necessary to perform a precise ICC analysis to identify the malicious flow of sensitive data. In this paper, we propose a static taint analysis method, named IccChecker, to identify the malicious ICC-based privacy leaks in Android applications. IccChecker first tracks the potential flow of sensitive data across components and extracts the contextual factors which trigger the sensitive behavior. By leveraging the context information, our approach differentiates the malicious privacy leaks from the legal privacy information exchanges according to the proposed contextual policy. Moreover, we present a comprehensive assessment with benchmarks and real-world applications. Our evaluation results with benchmarks demonstrate that IccChecker improves the precision of ICC-based privacy leak detection. In the evaluation with real-world applications, our approach identifies 4 apps with ICC-based privacy leaks among 168 Google Play apps (2.3%) while 31 apps are identified from 49 malwares (63.3%).**

I. INTRODUCTION

The smart phone has become an indispensable part of people's daily life in recent years. Among the different types of mobile operating systems, Android is extremely popular and possesses the largest market share currently. However, there also exist some serious privacy issues in the Android platform because of the numerous malicious and vulnerable apps (In the rest of the paper, apps are employed to denote the applications). As a result, user's privacy may be leaked carelessly or maliciously by these apps, which brings great security threats to the users.

Some static analysis approaches have been proposed [1]–[5] to mitigate the privacy leaks in Android apps. For example, Mann et al. [2] present a framework to identify the privacy leaks from the Android APIs by applying static information flow analysis techniques. Flowdroid [3], performs static taint analysis for Android apps by providing a precise model of the Android component lifecycle and callback methods. However, these prior works aim at detecting intra-component privacy leaks by static analysis, which operate within a single component. Unfortunately, there are privacy leaks which result from interactions across multiple components. In Android, developers usually intend to create rich and innovative apps by using a set of reusable components. Moreover, Android presents the ICC mechanism, which enables the apps to perform frequent interactions between different components. Also, the ICC mechanism can be maliciously used by malwares and cause the privacy data leakage problem.

To mitigate the threats caused by the ICC-based privacy leaks, many methods are proposed [6]–[14] to perform the ICC analysis. Epicc [9] parses the ICC parameters statically, but it has not considered to link the source and target components to perform the data flow analysis. DidFail [6] detects the ICC leaks by leveraging Epicc and static taint analysis [3]. However, the results give a higher false positive rate since it focuses on the ICC leaks among Activities via implicit Intents. Amandroid [11] and IccTA [12] are proposed to conduct the static taint analysis and detect the privacy leaks through the ICC channels in Android apps. Nevertheless, the taint flow of sensitive data across components may be legal or malicious, but they do not differentiate the legal sensitive flow from the malicious ICC privacy leaks. Since these existing static analysis approaches give the detection results with high false positive rates, false positive rate minimization is demanded to precisely identify the malicious ICC-based privacy leaks in Android apps.

In this paper, we propose IccChecker, a static ICC analysis tool, which detects the possible flow of sensitive data across components in Android apps to identify the malicious ICC-based privacy leaks. To improve the precision of the ICC analysis, we build links between the source and target components by adopting a probabilistic model. The path contexts, which trigger the ICC privacy leaks, are extracted by performing data-flow and control-flow analysis. By leveraging the context information, we propose a contextual policy to differentiate the malicious ICC privacy leaks from all possible flows of sensitive data.

Our contributions are as follows:

- We propose IccChecker, a static ICC analyzer to identify the ICC-based privacy leaks across different components in Android apps.
- We improve the precision of the ICC analysis by adopting

‡ Corresponding Author

978-1-5386-1629-1/17/$31.00 ©2017 IEEE

the probabilistic model to build ICC links.

- We propose a contextual policy to identify the malicious ICC privacy leaks according to the path contexts.
- Finally, we carry out a comprehensive assessment of IccChecker with the benchmarks and real-world apps. The results demonstrate that IccChecker can effectively identify the malicious ICC privacy leaks.

II. Background

In this section, we present an brief overview of the Android system and the motivation of the proposed work respectively.

A. Android Overview

1) Component: In Android, apps are created with a set of reusable components. Android OS provides four types of important components: *Activity*, *Service*, *Broadcast Receiver* and *Content Provider*. Activities dictate the UI and handle the interactions between the user and the smart phone. Services are usually used to handle the background processing associated with apps. Broadcast Receivers receive broadcast messages from other apps or the Android OS. Content Providers handle the data and database management issues. Android also present inter-component communication (ICC) mechanism, which allows different components to exchange data and invoke each other. Each component can be triggered by its embodying app or other apps through a set of Android ICC methods.

2) Intent: Different components can communicate with each other via sending Intents, which are used to request actions from other components. Intents can be explicit or implicit. Explicit Intents specify the app and class name of the target component, while the implicit Intents are usually used to activate the components in other apps and the fields for the target component name are left blank. Intents can be matched with their target component through Intent resolution process by the Android operating system at runtime. For static ICC analysis, the Intent need to be resolved statically to link the source and target component precisely.

3) Permission: To control the access to the privacy data and system functionalities of the mobile devices, Android employs a permission-based security mechanism. The corresponding permissions should be declared in the configuration file *AndroidManifest.xml* to perform sensitive operations. Besides, permission checks are performed to prevent the unauthorized access at runtime. By default, an Android app can only access a limited range of system resources.

B. Motivation

1) Privacy Related Behavior: A sensitive behavior can be described as an information flow path including a sequence of statements as follows:

$$Stmt_{source} \rightarrow Stmt_1 \rightarrow Stmt_2 \rightarrow ... \; Stmt_i \rightarrow Stmt_{sink}$$

Sources and sinks are methods which perform sensitive operations. $Stmt_i$ is the statement which the method i corresponds to. If the $Stmt_{source}$ is related with the sensitive data and protected by permissions, and the sensitive data is sent out of the app or device by the $Stmt_{sink}$, this behavior is defined as privacy-related, which may occur within a single component or across multiple components. The ICC-based privacy-related behavior, which occurs across multiple components, is defined as a special path which contains at least one ICC method in its statement sequence.

Listing 1. A Motivating Example

```
1  //class mainActivity
2  void onCreate(View v) {
3    TelephonyManager telMng = (TelephonyManager)
4     getSystemService(Context.TELEPHONY_SERVICE);
5    String id = telMng.getDeviceId();
6    Intent i1 = new Intent(mainActivity.this,
7        normalActivity.class);
8    i1.putExtra("deviceId", id);
9    startActivity(i1);
10   Date date = new Date();
11   Intent i2  = new Intent(mainActivity.this,
12       sendSMSservice.class);
13   if(date.getHours()>23 || date.getHours()< 5){
14    i2.putExtra("deviceId", id);
15    startService(i2);}
16   ...
17 }
18 //class normalActivity
19 void onStart() {
20   //legal behaviors
21  ...
22 }
23 //class sendSMSservice
24 int onStartCommand(Intent i, int flags,
25     int startId) {
26   String id=i.getExtras().getString("deviceId");
27   SmsManager smsMng = SmsManager.getDefault();
28   smsMng.sendTextMessage(number, null, id,
29       null, null);
30  ...
31 }
```

A concrete example is given in Listing 1. The *mainActivity* component obtains the device ID (line 5) which is encapsulated in an Intent and passed to the *normalActivity* component by calling the ICC method *startActivity()*. Besides, the *mainActivity* passes the sensitive data to the *sendSMSservice* component by calling the ICC method *startService()* if the time is between 11 pm and 5 am. Then, the *onCreate()* of *sendSMSservice* calls *SmsManager.sendTextMessage()*, which is considered as a sink, to send the sensitive data out of the device. Obviously, the behavior from obtaining the sensitive data in line 5 to sending the data out of the device in line 28 is an ICC-based privacy-related behavior.

2) Malicious ICC Privacy Leak: The apps with sensitive behaviors may be legal, i.e., the operation of sending the privacy data out of the app or device may be required by the apps to realize their functionalities. However, the privacy data may also be sent out deliberately by the malicious apps, which is obviously privacy leaks. For example, the behavior of uploading the GPS coordinates from one device to a third party is good in Google Map and malicious in other malwares. In this paper, the malicious privacy leaks, which occur during the interactions among multiple components, is aimed to be identified.

III. Contextual Factors

For apps, the execution of their behaviors depends on the contextual factors. For example, the user privacy will be sent out of the devices only if some system events are triggered

or the environment conditions are satisfied. In the motivating example, the source method *getDeviceId()* is activated by a lifecycle method *onCreate()* which is considered as an event. In the component *sendSMSservice*, the sensitive data is sent out of the device by the sink method *sendTextMessage()*. Besides, a critical conditional statement needs to be satisfied that the system time should be in the specific time period in the example. The system time is obtained by invoking the interface *getHours()*, which is an environment attribute *Calendar information* and useful to analyze the anomalies in apps. In this paper, these contextual factors on the path of the sensitive behavior are used to identify the malicious ICC privacy leak. In this section, details of the considered contextual factors are introduced.

A. Considered Permissions

The corresponding permissions need to be requested by Android apps to perform sensitive operations. To have a better understanding of the permission requests in malware, it is necessary to present the top used privacy-related permissions in malicious apps. To analyze the permissions, we use the malware dataset Drebin [15], which is one of the largest and newest datasets of Android malwares. As shown in Figure 1, the top 25 permissions are presented from the statistical permissions in Drebin dataset.

Android predefines specific permissions to protect the system sensitive resources, including the privacy data and system functionalities. A permission-protected method may not be privacy-related. For example, although the popularly used method *ITelephony.call()* is protected by the permission *android.permission.CALL_PHONE*, it is a sensitive method but not privacy-related. The previous work SUSI [16] specfies a detailed list of sources *sources(SUSI)* and sinks *sinks(SUSI)* for information flows. In addition, a detailed list of API calls and corresponding permission mappings is provided by PScout [17]. In this paper, the set of methods $M_{PScout}(p)$= $\{m_1, m_2, ..., m_n\}$, where p is the corresponding permission, and m_i is the mapping method in PScout. If $m_i \in$ *sources(SUSI)*, the corresponding permission p is considered as privacy-related.

B. Privacy Related Method

A privacy-related behavior can be described as a path $path_{privacy}$*(source, sink)*, which consists of a source from which the privacy data originate and a sink to which the data is sent. These sources and sinks are considered as the privacy-related methods in this paper.

The methods specified in *sinks(SUSI)* are used as our privacy-related sink methods, i.e., $M_{sink} = \{(m) \mid m \in$ *sinks(SUSI)*$\}$. The sinks can be either protected by permission or not. For example, *FileOutputStream.write()* is not a permission protected method but is a sink method in *sinks(SUSI)* to write the data to a file. Since we focus on the privacy-related behaviors in Android apps, the methods, which access the sensitive resources and are protected by permissions, are selected as our sources. The top 25 privacy-related permissions

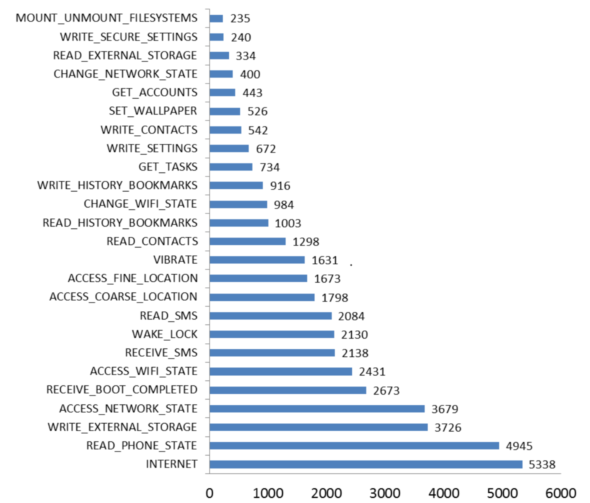

Figure 1. Top 25 Permissions in Drebin Dataset

in Figure 1 are represented as a set *S(p)*. The set of privacy-related source methods is represented as $M_{source}(p) = \{(p,m) \mid p \in S(p)$*, and* $m \in M_{PScout}(p)$ *and* $m \in$ *sources(SUSI)*$\}$.

C. Mapping Feature

Apps usually requests multiple permissions to satisfy different functionalities, while each permission is defined to protect multiple sensitive methods. Besides, each method may be used in different segment of the code, which is described as different statements $Stmt_i$. Moreover, the sinks in the information flow paths can be classified into several different categories. In this paper, the relationships among the permission, source, $Stmt_{source}$ and sink category are defined as the mapping features.

To illustrate the mapping feature, *com.kniusw.phoneloc* is selected as an example from the Drebin dataset. As shown in Figure 2, the permission *READ_PHONE_STATE* is used to protect several methods such as *getDeviceId(), getSubscriberId()* and *getSimCountryIso()*. The statement *$r6 = virtualinvoke $r5.<android.telephony.TelephonyManager: java.lang.String getSubscriberId()>()* and *$r1 = virtualinvoke $r4.<android.telephony.TelephonyManager: java.lang.String getSubscriberId()>()* are different statements which correspond to the identical method *getSubscriberId()*. The sink method *Log.e()* is classified as the *LOG* category while the method *SmsManager.sendTextMessage()* belongs to the *MESSAGE* category. Usually, one permission is requested by the developers for a specified functionality in an app. Intuitively, the following three features are potentially malicious in Android apps.

- F_1: For one permission, some mapping methods correspond to $path_{privacy}$*(source, sink)*, while the others not.
- F_2: For one method, it is invoked multiple times at different segments, which can be described as different source statements. Some source statements have sinks while the others not.
- F_3: For one source, it has different sinks, which belong to the different categories.

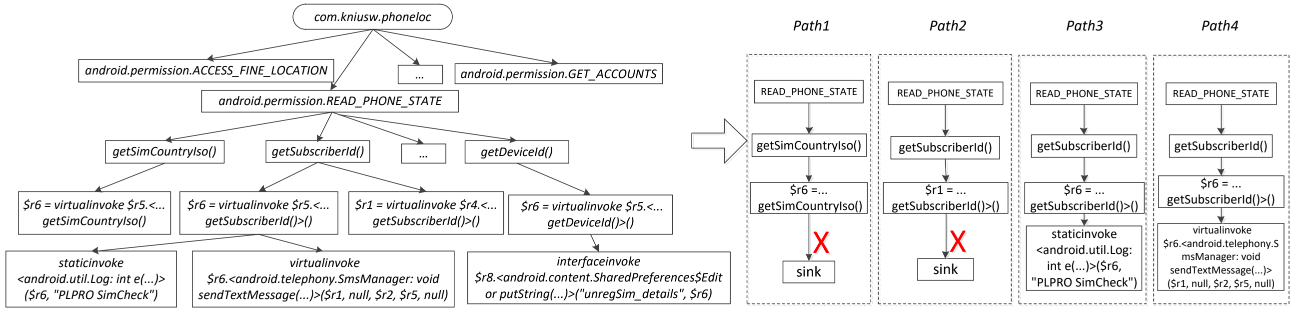

Figure 2. Mapping Feature

Table I
MAPPING FEATURES IN REAL-WORLD APPS

Dataset	Total	F1	F2	F3
Drebin	100	59 (59%)	58 (58%)	43 (43%)
Google Play	100	20 (20%)	17 (17%)	3 (3%)

There exist four paths in Figure 2. Feature 1 can be described as F_1: $\exists$ *path 1* $\cap$ *path 3*. Feature 2 can be described as F_2: $\exists$ *path 2* $\cap$ *path 3*. Feature 3 can be described as F_3: $\exists$ *path 3* $\cap$ *path 4*, and the *sinks* belong to different categories.

To the best of our knowledge, there is no investigation of the mapping features in Android apps. To propose the mapping features, we perform a study on real-world apps. We selected 100 apps from Google Play and 100 apps from the Drebin malwares randomly as our dataset. Table I shows the number of apps with the mapping features introduced above. The results demonstrate that the apps with the three features in malware dataset significantly outnumber the apps from Google Play. Since Google Play is the official app store for Android, the apps from Google Play is considered as benign. Therefore, the three features can contribute to differentiate the malicious apps from the benign ones.

D. Activation Event

In Android, a privacy-related behavior can be triggered in several ways. The graphical user interfaces in an app can trigger the security-sensitive behavior, e.g., clicking a button in the app. Besides, an app can be initiated by the Android system, i.e, the app code is called by Android APIs such as receiving the broadcast and calling the lifecycle method. In addition, the interactions on the device interfaces can also trigger the privacy-related behavior, e.g., pressing the HOME or BACK button.

The activation event is defined as the event which leads to a privacy-related method call in an app. The activation events can be categorized into two categories: *UI event* and *nonUI event*. The events, which trigger the privacy-related behavior by user interactions (via app or device interfaces), are defined as *UI events*, while others are defined as *nonUI events*.

E. Environment Attribute

There exist many conditional statements in an app code and the control flows of the privacy-related method calls can be affected by the values of these conditional statements. In malware, there will be more conditional statements to hide the malicious intent. The behavior, which contains the environmental attribute, is potentially malicious. Inspired by AppContext [18], the environment attribute in conditional statement is considered as a contextual factors in this paper.

IV. OUR APPROACH

In this section, we introduce our approach IccChecker, which is designed to identify the malicious ICC-based privacy leaks by using the static taint analysis and contextual policy. Figure 3 illustrates the pipeline of IccChecker. Since this paper addresses the privacy leaks which result from the interactions among multiple components, the ICC values in different components are extracted firstly. Then, links between the source and target components are built according to the parsed ICC values. By leveraging the ICC links, different components are connected to perform the ICC-based static taint analysis via code instrumentation technique. To extract the contextual factors of the privacy-related behaviors, the Android bytecode is instrumented and an inter-procedure control flow graph (ICFG) [19] is obtained. Finally, the malicious ICC privacy leaks are identified by leveraging the results of the ICC-based static taint analysis and the contextual factors.

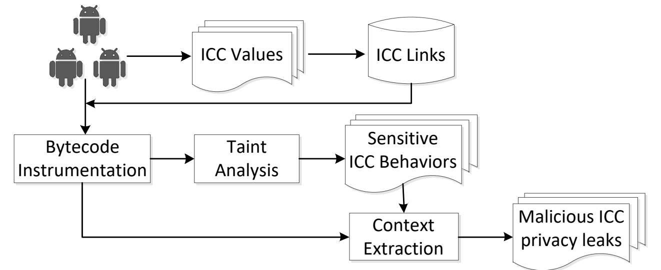

Figure 3. Pipeline of IccChecker

A. Privacy Related ICC Behavior Extraction

In this part, we detail our approach to build the ICC links across different components in the analyzed apps. We further describe the method to extract the privacy-related behavior by performing taint flow analysis according to the ICC links.

1) ICC Links: An ICC link is defined as a link between a source component and a target component. A source usually contains an ICC method which sends Intent to the target. The Intent describes the characteristics of the target component. For example, there are two ICC links in the motivating example. One is *mainActivity* $\rightarrow$ *normalActivity* and the other

is *mainActivity* → *sendSMSservice*. To match the components which communicate with each other, Android system performs Intent resolution process according to the Intent values at runtime. To infer the links effectively in static analysis, the ICC values (e.g., ICC method, Intent) in the analyzed apps should be parsed precisely. In our approach, IccChecker uses the ICC values computed by IC3 [10], which presents a state-of-the-art approach to fully parse the data field of Intents.

The computed Intents should be matched with the potential target components to build the ICC links. Since the matching precision is directly related to the precision of the ICC analysis, the probabilistic model of ICC in PRIMO [13] is adopted to accurately build the ICC links according to the static analysis results in the analyzed apps.

2) Taint Flow Analysis for ICC: Different from other Java-based programs, there are discontinuities in Android apps due to the existence of the lifecycle methods, callback methods and ICC methods. Android apps are component-based, where each component possesses a series of lifecycle methods (e.g., *onCreate()*) and callbacks methods (e.g., *onClick()*). Besides, frequent interactions between different components within an app or across multiple apps are performed by ICC methods (e.g., *startActivity()*). There is no direct call among these three types of methods in the code of apps since these methods are handled by the Android system. To perform the taint flow analysis for ICC precisely, the static analysis must consider these discontinuities.

IccChecker performs a taint flow analysis by adopting Flow-Droid [3], which provides a precise modeling of the Android component lifecycle and callback methods. To implement the propagation of taint flow between different components, the ICC links we built and the instrument approach presented in IccTA [12] are employed to connect different components. Finally, a set of $path_{privacy}$*(source, sink)* can be obtained.

B. Context-based ICC Privacy Leak Detection

According to the given sources and sinks, all the privacy-related ICC behaviors will be extracted, which can be legal or malicious. To effectively identify the malicious apps that leak privacy data by ICC, the app analysis should differentiate the malicious data flows from the legal ones. In this paper, we adopt the contextual factors to identify the malicious ICC privacy leaks.

1) Path Context Extraction: Given a $path_{privacy}$*(source, sink)*, the privacy-related methods API_{source} and the API_{sink} can be obtained according to the results of Flowdroid. We obtain the corresponding permissions $Perm_{source}$ and $Perm_{sink}$ by matching the APIs and permissions provided by PScout.

Algorithm 1 presents the process of extracting the mapping features for a given privacy-related path. The algorithm takes the information flow results from Flowdroid as the inputs and returns the three features. All the privacy-related methods and the corresponding statements are recorded by IccChecker. By comparing the methods/statements in the results of Flowdroid and the records of IccChecker, the F_1 and F_2 can be extracted. F_3 can be extracted by the statistical analysis of the inputs.

Algorithm 1 Extract Mapping Features

```
Input  : Results: Information flow results of Flowdroid
Output: F1, F2, F3: three mapping features
 1: F1, F2, F3 ← false
 2: usedMethods ← getUsedMethods(results)
 3: for source ∈ results do
 4:     leakMethods.add(getMethod(source))
 5:     leakStmts.add (getStmt(source))
 6: end for
 7: for path_privacy(source, sink) do
 8:     Perm_source ← getPermission(source)
 9:     // all privacy-related methods used in app codes
10:     methods ← getMethods(Perm_source)
11:     for m ∈ methods do
12:         if m ∉ leakmethods then
13:             F1 = true
14:         end if
15:     end for
16:     API_source ← getMethod(source)
17:     // all API_source related statements in app codes
18:     statements ← getStatements(API_source)
19:     for stmt ∈ statements do
20:         if stmt ∉ leakStmts then
21:             F2 = true
22:         end if
23:     end for
24:     sinks ← getSinks(source)
25:     for sink ∈ sinks do
26:         sinkCategories.add(getCategory(sink))
27:     end for
28:     if sinks.size > 1 and sinkCategories.size > 1  then
29:         F3 = true
30:     end if
31: end for
32: return  F1, F2, F3
```

For a given $path_{privacy}$*(source, sink)*, the mapping feature $F_{mapping} = F_1 \cup F_2 \cup F_3$.

The activation events are divided into *UI event* and *nonUI event* according to the entry points. All the entry points of the privacy-related method are recorded and classified by IccChecker according to the involvement of user interactions.

The control flows from the entry points to the privacy-related method calls can be affected by the conditional statements with environmental attributes. To precisely extract these environmental attributes, we adopt the reduced inter-procedure control flow graph (RICFG) from AppContext [18], which contains all the paths from an entry point to a privacy sensitive method call. Given a privacy method call and the corresponding activation event, a set of environmental attributes of conditional statements are saved.

2) Identifying Malicious ICC privacy Leaks: By leveraging the extracted path contexts, we propose a contextual policy to differentiate the malicious and benign privacy-related ICC behaviors. The contextual policy is presented in Algorithm 2. The process takes the contextual factors of $path_{privacy}$*(source, sink)* as inputs and outputs the malicious behavior.

Since this paper focuses on the privacy-related behaviors, only the privacy-related path $path_{privacy}$*(source, sink)*, where the source is protected by permission, is considered as the potential privacy leaks, i.e., once the $Perm_{source}$ is null, the path is classified as benign. Otherwise, if the $F_{mapping}$ is true, i.e., there exists at least one feature, the path is considered

Algorithm 2 Identify Malicious ICC Leak

Input : Contextual factors of $path_{privacy}$*(source, sink)*
Output: Malicious or Benign
1: *malicious* ← ***false***
2: **if** $perm_{source}$ ≠ *null* **then**
3: **if** $F_{mapping}$ = ***true*** **then**
4: *malicious* ← ***true***
5: **return** $malicious$
6: **end if**
7: // all activation events of source and sink methods
8: *actEvents* ← *getActEvents()*
9: **for** $actEvent$ ∈ actEvents **do**
10: **if** $actEvent \in nonUI_{event}$ **then**
11: // all conditional statements of an activation event
12: *conditions* ← *getConditions(*Act_{event}*)*
13: **for** $Stmt_{con}$ ∈ conditions **do**
14: **if** $Stmt_{con} \in attrEnvs$ **then**
15: *malicious* ← ***true***
16: **end if**
17: **end for**
18: **end if**
19: **end for**
20: **end if**
21: **return** $malicious$

as a potentially malicious privacy leak. In the algorithm, $actEvents$ represents all the activation events of the source and sink methods and $nonUI_{event}$ represents the set of *nonUI events*. The privacy-related behaviors are considered occurring with the user interactions and classified as benign while the $actEvent \notin nonUI_{event}$. Otherwise, if the conditional statement *Stmt*$_{con}$ of the corresponding activation event belongs to the environmental attributes $attrEnvs$, the sensitive path is classified as malicious. The malicious privacy-related ICC behavior represents an ICC privacy leak while the benign one represents a legal behavior.

V. Evaluation

To evaluate the effectiveness of IccChecker, two evaluations are conducted. First, IccChecker is compared with an existing tool IccTA [12]. Then, the effectiveness of IccChecker is evaluated by identifying the malicious ICC privacy leaks in real-world apps.

A. Comparison with Existing Tools

The precision of ICC privacy leaks is decisive to the effectiveness of distinguishing the malicious ones. To evaluate the of precision of ICC privacy leak, we compare our IccChecker with the existing tool IccTA [12], which is the most recent state-of-the-art tool to detect the ICC leaks. Since the evaluations in [12] illustrates that IccTA outperforms the existing tools by achieving a better precision and recall, we just perform comparisons with IccTA.

1) Benchmarks: To compare our IccChecker with IccTA, we employ 31 test cases from two benchmarks: DroidBench and ICCBench. DroidBench is provided by the FlowDroid [3] for evaluating the information-flow analysis. ICCBench is another benchmark introduced by Amandroid [11].

2) ICC Leak Test: Table II presents the results in the detection of the ICC leaks. For the 31 test cases, IccChecker achieves a precision of 100% and a recall of 87.5% while

Table II
TEST RESULTS ON BENCHMARKS
O = True Positive, * = False Positive, X = False Negative.

Package Name	IccTA	IccChecker
DroidBench		
edu.mit.icc_action_string_operations	O *	O
edu.mit.icc_componentname_class_constant	O	O
edu.mit.icc_concat_action_string	O *	O
edu.mit.icc_intent_component_name	O	O
edu.mit.icc_intent_passed_through_api	X	X
edu.mit.icc_non_constant_class_object	O	O
edu.mit.icc_pass_action_string_through_api	O *	O
edu.mit.icc_broadcast_programmatic_intentfilter	O	O
edu.mit.icc_component_not_in_manifest		
edu.mit.icc_event_ordering	X	X
lu.uni.snt.serval	X	X
edu.mit.to_components_share_memory	O O	O O
ICCBench		
com.ksu.passwordPassTest	O	O
com.ksu.fieldFlowSentivity		
com.ksu.explicit_nosrc_nosink		
com.ksu.explicit_nosrc_sink		
com.ksu.explicit_src_nosink		
com.ksu.explicit_src_sink	O	O
com.ksu.implicit_nosrc_nosink		
com.ksu.implicit_nosrc_sink		
com.ksu.implicit_src_nosink		
com.ksu.implicit_src_sink	O	O
com.ksu.dynamicregister1	O	O
com.ksu.dynamicregister2	O	O
com.ksu.explicit1	O	O
com.ksu.implicit1	O	O
com.ksu.implicit2	O	O
com.ksu.implicit3	O	O
com.ksu.implicit4	O	O
com.ksu.implicit5	O	O
com.ksu.implicit6	O	O
Sum, Precision , Recall and F-Measure		
O, higher is better	21	21
*, lower is better	3	0
X, lower is better	3	3
Precision p = O/(O + *)	87.5%	100%
Recall r = O/(O + X)	87.5%	87.5%
F-measure 2pr/(p + r)	0.875	0.93

the precision and recall of IccTA are both 87.5%. There are three false alarmed leaks in IccTA, which are all caused by the false links, i.e., the Intent is pointed to a wrong target component. However, the source and target component are linked precisely in our IccChecker. Since the false negatives depend on the ICC values computed by IC3 [10], IccChecker and IccTA give the same amount of false negatives. However, IccChecker outperforms IccTA with a better precision since we adopt a probabilistic model to reduce the false positive rate of ICC links.

B. Effectiveness

To evaluate the effectiveness of IccChecker, we run IcCChecker on real-world apps and present the experimental results by giving the identified malicious ICC privacy leaks.

1) Dataset Collection: Two distinct datasets are employed in this evaluation. The first dataset is MalGenome which is collected by Zhou et al. [20] and includes 1,260 Android malware samples. The second dataset includes 500 Google Play apps which are collected from Google market and considered as benign apps. Since we focus on the ICC-based privacy-related behavior in this paper, these apps are preprocessed. We first

Table III
TEST RESULTS ON REAL-WORLD APPS

Dataset	the Number of Apps			the Number of Behaviors		
	senIccBehavior	priIccBehavior	malIccBehavior	senSum	priSum	malSum
MalGenome	49	32	31 (63.3%)	1145	231	168 (14.6%)
GooglePlay	168	13	4 (2.3%)	4117	99	17 (4.1%)

perform a permission filter to select the apps which contains at least on of the top 25 permissions and the corresponding APIs. Then we exclude the apps without the ICC methods.

After the preprocessing, we further filter out the apps that throw exceptions or timeout caused by FlowDroid and IC3. The final dataset contains 217 apps in total, including 168 benign apps and 49 malicious apps. We apply IccChecker to these apps to identify the malicious ICC privacy leaks.

2) Experimental Results: The results are shown in Table III. For MalGenome, 49 apps are reported containing at least one sensitive behavior across components, with a total of 1145 ICC-based sensitive behaviors. Among the apps with sensitive ICC behaviors, 32 apps are reported containing at least one privacy-related behavior by ICC, with a total of 231 privacy paths. 31 apps are identified by IccChecker that they contain at least one malicious ICC-based privacy leak, with a total of 168 privacy paths, i.e., 168 malicious ICC privacy leaks are identified among 1145 ICC-based sensitive behaviors (or 14.6%) in the malware dataset. For apps from Google Play, 168 apps are reported containing at least one sensitive behavior across components, with a total of 4117 ICC-based sensitive behaviors. 13 apps are reported containing at least one privacy-related behavior by ICC, with a total of 99 privacy leaks. IccChecker identifies 4 apps containing at least one ICC-based privacy leak, with a total of 17 privacy leaks. In other words, 17 malicious ICC privacy leaks are identified among 4117 ICC-based sensitive behaviors (or 4.1%) in Google Play apps. The results in Table III demonstrate that the potential malicous apps with ICC privacy leaks identified by IccChecker in malware dataset (63.3%) are significantly more than that in Google Play dataset (2.3%).

VI. DISSCUSSION

In this paper, we detect the ICC-based privacy leak behavior. However, the malicious ICC behaviors may not leak any information, e.g., making expensive calls to a specific phone number. We will consider various of ICC behaviors and explore to detect malicious ones in the future work. Besides, IccChecker is used to identify the malicious ICC privacy leaks in single app currently, the leaks resulting from the inter-app communication (IAC) will also be considered in the future work. Moreover, due to the lack of ground truth for determining a security-sensitive method call to be malicious or benign, we have not presented the false negatives and false positives in our experiments. We will determine and label the malicious security-sensitive method calls manually to evaluate the effectiveness of IccChecker in the future work.

VII. RELATED WORK

Many threats have been presented in Android due to the interactions between components. The ICC problems have received numerous attentions in recent years.

To perform ICC analysis precisely, some tools have been developed [9], [10], [13], [21]. By identifying ICC methods as well as their parameter values (e.g., action, category), Epicc [9] is proposed to detect all the potential Intent-based communication channels between different application components. IC3 [10] develops an advanced tool which implements the idea of Epicc. It also parses the URIs (e.g., scheme, host) to support Content Provider related ICC methods (e.g., query) and to fully support the data field of Intents. IC3 further infers the ICC values, which are required for understanding the interactions among the components of Android apps. To rank the ICC links based on the likelihood that they are actual linked, PRIMO [13] overlays a probabilistic model of ICC on top of the static analysis results by leveraging IC3. PRIMO aims to perform triage on these links, with the goal of prioritizing the true positives over the false positives.

Multiple prior works use ICC analysis to mitigate the threats caused by the ICC-based privacy leaks [6], [11], [12], [14]. Amandroid [11] and IccTA [12] are closely related to IccChecker which aim at detecting the ICC-based privacy leaks in Android apps. Amandroid conducts a static analysis across the components to detect the privacy leaks. However, the Content Provider, one of the four Android components, is not tackled in Amandroid. Amandroid is also insensitive to some complicated ICC methods such as *bindService()* and *startActivityForResult()*. Compared to Amandroid, IccTA improves the analysis by building ICC links and compute the taint paths. However, IccTA produces false positives in the ICC link building. Besides, IccTA reports all discovered flows from sources to sinks and does not differentiate the malicious and benign paths. Therefore, we adopt PRIMO, which overlays a probabilistic model of ICC on top of the static analysis results, to infer the ICC links more accurately. We further extract the context information on the path to identify the malicious paths.

Several approaches are proposed to detect intra-component privacy leaks in Android apps [1]–[5]. Nevertheless, it focuses on the single component. Some other tools have been developed to detect the component vulnerabilities in Android apps. CHEX [8] is a static analysis tool to automatically vet Android apps for component hijacking vulnerabilities by linking pieces of code reachable from entry points. Although CHEX can discover data flows between the Android application components, it does not address the data flow through ICC. ComDroid [7], which is used to analyze apps before release via static analysis, studies the security challenges in Android communications and

focuses on the Intent related issues. Unfortunately, it is coarse-grained and gives false positives. PCLeaks [22] performs the data-flow analysis to detect potential component leaks, which includes the component hijacking vulnerabilities and the component injection vulnerabilities. ContentScope [23] targets at examining the vulnerabilities of an unprotected Content Provider. PCLeaks and ContentScope are static analysis tools which focus on vulnerability mining and analyzing. Nevertheless, we focus on the taint data to detect the privacy leaks across multiple components in this paper.

VIII. Conclusion

To identify the malicious privacy leaks among the components in Android apps, we propose IccChecker, a static taint analysis tool. IccChecker extracts the ICC behaviors by building the ICC links and performing the taint analysis. By leveraging the path contexts, we propose a contextual policy to perform the malicious ICC privay leak detection. IccChecker improves the precision of ICC analysis and differentiates the malicious ICCs with the legal ones. We present a comprehensive assessment to the proposed approach with benchmarks and real-world apps. Compared with IccTA, the evaluation results demonstrate that IccChecker is more effective in finding the sensitive ICC paths. The assessments with real-world apps indicate that the potential malicous apps with ICC privacy leaks identified by IccChecker in malware dataset (63.3%) are significantly more than that in Google Play dataset (2.3%).

Acknowledgment

This work was supported by National Key Research and Development Plan (No.2016YFB0800403), National Natural Science Foundation of China (No. 61402477, 61422213, U1605252), Beijing Natural Science Foundation (4172068), Key Program of the Chinese Academy of Sciences (No. QYZDB-SSW-JSC003).

References

[1] C. Gibler, J. Crussell, J. Erickson, and H. Chen, “Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale,” in *Trust and Trustworthy Computing - 5th International Conference (TRUST’12), Vienna, Austria, Jun. 2012*, pp. 291–307.

[2] C. Mann and A. Starostin, “A framework for static detection of privacy leaks in android applications,” in *Proceedings of the ACM Symposium on Applied Computing (SAC’12), Riva, Trento, Italy, Mar. 2012*, pp. 1457–1462.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, “Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14), Edinburgh, United Kingdom,Jun. 2014*, p. 29.

[4] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, “Scandal: Static analyzer for detecting privacy leaks in android applications,” *MoST*, vol. 12, 2012.

[5] Z. Yang and M. Yang, “Leakminer: Detect information leakage on android with static taint analysis,” in *Software Engineering (WCSE), 2012 Third World Congress on*. IEEE, 2012, pp. 101–104.

[6] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, “Android taint flow analysis for app sets,” in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis (SOAP’14), Edinburgh, UK, Jun. 2014*, pp. 5:1–5:6.

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, “Analyzing inter-application communication in android,” in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys ’11), Bethesda, Maryland, USA, 2011*, pp. 239–252.

[8] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: statically vetting android apps for component hijacking vulnerabilities,” in *the ACM Conference on Computer and Communications Security (CCS’12)*, Raleigh, NC, USA, Oct. 2012, pp. 229–240.

[9] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, “Effective inter-component communication mapping in android: An essential step towards holistic security analysis,” in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, Aug. 2013*, pp. 543–558.

[10] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, “Composite constant propagation: Application to android inter-component communication analysis,” in *37th IEEE/ACM International Conference on Software Engineering (ICSE’15), Florence, Italy, May. 2015*, pp. 77–88.

[11] 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 *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, Nov. 2014*, pp. 1329–1341.

[12] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, “Iccta: Detecting inter-component privacy leaks in android apps,” in *37th IEEE/ACM International Conference on Software Engineering (ICSE’15), Florence, Italy, May. 2015*, pp. 280–291.

[13] D. Octeau, S. Jha, M. Dering, P. D. McDaniel, A. Bartel, L. Li, J. Klein, and Y. L. Traon, “Combining static analysis with probabilistic models to enable market-scale android inter-component analysis,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’16), St. Petersburg, FL, USA, Jan. 2016*, pp. 469–484.

[14] X. Cui, D. Yu, P. P. F. Chan, L. C. K. Hui, S. Yiu, and S. Qing, “Cochecker: Detecting capability and sensitive data leaks from component chains in android,” in *Information Security and Privacy - 19th Australasian Conference, (ACISP’14), Wollongong, NSW, Australia, Jul. 2014*, pp. 446–453.

[15] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, “DREBIN: effective and explainable detection of android malware in your pocket,” in *21st Annual Network and Distributed System Security Symposium (NDSS’14), San Diego, California, USA, February 23-26, 2014*.

[16] S. Rasthofer, S. Arzt, and E. Bodden, “A machine-learning approach for classifying and categorizing android sources and sinks,” in *21st Annual Network and Distributed System Security Symposium (NDSS’14), San Diego, California, USA, Feb. 2014*.

[17] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “Pscout: analyzing the android permission specification,” in *the ACM Conference on Computer and Communications Security (CCS’12), Raleigh, NC, USA, Oct. 2012*, pp. 217–228.

[18] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, “Appcontext: Differentiating malicious and benign mobile app behaviors using context,” in *37th IEEE/ACM International Conference on Software Engineering (ICSE’15), Florence, Italy, May. 2015, Volume 1*, pp. 303–313.

[19] M. J. Harrold, G. Rothermel, and S. Sinha, “Computation of interprocedural control dependence,” in *ACM SIGSOFT Software Engineering Notes*, 1998, pp. 11–20.

[20] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *IEEE Symposium on Security and Privacy (SP’12), San Francisco, California, USA, May. 2012*, pp. 95–109.

[21] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, “Scandroid: Automated security certification of android,” Tech. Rep., 2009.

[22] L. Li, A. Bartel, J. Klein, and Y. L. Traon, “Automatically exploiting potential component leaks in android applications,” in *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom’14), Beijing, China, Sep. 2014*, pp. 388–397.

[23] Y. Zhou and X. Jiang, “Detecting passive content leaks and pollution in android applications,” in *20th Annual Network and Distributed System Security Symposium (NDSS’13), San Diego, California, USA, Feb. 2013*.