IacCE: Extended Taint Path Guided Dynamic Analysis of Android Inter-App Data Leakage

Tianjun Wu(✉) and Yuexiang Yang

College of Computer, National University of Defense Technology,
Changsha 410073, China
{wutianjun08,yyx}@nudt.edu.cn

Abstract. There exists a need for overall security analysis of a set of apps. We demonstrate IacCE, a tool implementing our approach that applies concolic execution on combined apps guided by extended Inter-App taint paths. Furthermore, we replay the event-and-input generated by concolic execution on the original app set, to monitor the actual data-leakage behavior. To our knowledge, we are the first to apply concolic execution for dynamic analysis of Inter-App communications.

Keywords: Inter-App communication · Android application · Dynamic analysis · Concolic execution · Static taint analysis

1 Introduction

Android is by far the most ubiquitous mobile operating system, and we are witnessing a surge in the adoption of Android applications (also called *apps*). The situation for app security is severe, due to the weakness of the permission system and the programming model. The Inter-Component communication model for instance, which is used as an efficient data-exchange mechanism for loosely-coupled apps, might be misused to leak private data outside the device without user consent.

There are growing efforts for analyzing Android apps, aimed at discovering such safety issues as malware behavior and application vulnerabilities. Static app analysis tools, such as the static taint analysis tool FlowDroid [4], can efficiently analyze all the code in the application, but they are inherently imprecise as there may be behavior misses or falsely behavior report. Dynamic analysis tools, such as TaintDroid [29], avoid those shortcomings, but are relatively slow as they have to run the code, and are inherently incomplete as they can only tell the behavior that they execute [18–20].

Despite those researches on single app security, there are few tools for Inter-App vulnerability analysis. Literature [3] performed an investigation on 500 apps from Google Play [24], F-Droid [25], Bazaar [26], and MalGenome [27]. It found that only 32 percent of acquired permissions are necessary for API calls and averagely each app has about two unchecked but used permissions. This incurs a vulnerable path from the exported interface of the app component to the API use, which can be exploited by the interaction of the app with other apps. Issues related to this kind of vulnerability already exist, such as collusion attacks and privilege escalation chaining [28]. The need for overall security analysis of a set of apps exists.

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 317–333, 2017.
DOI: 10.1007/978-3-319-59608-2_19

Thus, in this paper, we propose the first tool named "IacCE" (analyzing Inter-App Communications using Concolic Execution) that dynamically analyze Inter-App data leakage by combining static taint analysis and cocolic execution. We first apply static Inter-App data-flow analysis on the combined app of the app set, then generate inputs and events to execute sensitive Inter-App paths by extended-taint-path guided concolic execution, finally dynamically verify the leakage by executing apps in the app set with those generated inputs and events.

The contribution of this paper is three fold. First, to our knowledge, we are the first to apply concolic execution for dynamic analysis of Inter-App communications. Our combination of static taint analysis and concolic execution achieves higher precision and recall than state-of-the-art tools. Second, we developed IacCE, an open-source tool for Inter-Component and Inter-App dynamic analysis. Third, we compose a benchmark based on DroidBench [22] and ICC-Bench [23] for better assessment of Inter-Component and Inter-App analyzers with 77 apps.

2 Background

Android Basis. Android defines four types of app component, i.e., *Activity* (defining user interface), *Service* (performing background processing), *ContentProvider* (managing database), and *BroadcastReceiver* (receiving Inter-App broadcast messages). There are discontinuities within a component, which are used to drive apps with runtime *events* (system events or user interactions) and *life-cycle callbacks* (state transition of an app) from Android framework, besides the traditional input form of *data inputs*. Android provides specific methods, for triggering Inter-Component communications (*ICC*) and Inter-App communications (*IAC*). These methods are called with *Intent*, which specifies the *action*, *category*, *mimetype*, *data*, etc. Intent can be either *explicit* or *implicit* by define the receiver component or not. Components determine which Intent to receive by specifying an *Intent Filter*. Android permission system identifies the privileges of an app in the manifest file.

Static Taint Analysis. Static taint analysis starts at a sensitive source (location get by *getLastKnownLocation()*, for instance) and then tracks the sensitive data through the app until it reaches a sensitive sink (e.g. the *sendTextMessage()* API) [4]. It gives precise information about which data may be leaked.

Concolic Execution. *Concolic* (concrete + symbolic) *execution* (or dynamic symbolic execution) uses a combination of concrete and symbolic execution to analyze how input values flow through a program as it executes, and uses this analysis to identify other inputs that can result in alternative execution behaviors [10]. It traces symbolic registers at each conditional statement in order to build path conditions for specific execution traces. After collecting path constraints, a constraint solver is used for solving them and the result is just the program input we desire.

3 Motivating Example

To motivate and illustrate our approach, consider the app set "SWE" (SendSMS, WriteFile, and Echoer) in Fig. 1, which leaks data through Inter-App communication. The apps are inspired those used by IccTA [1] and DidFail [2], but further contain several challenging issues for existing static analysis methods as described in [3].

(A)

```
public class SendSMS extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        Button b = (Button) findViewById(R.id.b);
        b.setOnClickListener(new OnClickListener(){
            public void onClick(View v) {
                Intent i = new Intent(Intent.ACTION_SEND);
                i.setType("text/plain");
                String uid = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE).getDeviceId();// SRC
                StringBuilder sb = new StringBuilder();
                sb.append("secret");
                sb.append("1");// SRC
                i.putExtra(sb.toString(), uid);
                this.startActivityForResult(i, 0);// SNK
            }
        });
    }

    protected void onActivityResult(int rq, int rs, Intent i) {
        ...
        String msg = i.getExtras().getString("secret1");// SRC
        SmsManager.getDefault().sendTextMessage("10086",, msg,,);// SNK
        ...
    }
}
```

(B)

```
public class WriteFile extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        Button b = (Button) findViewById(R.id.b);
        b.setOnClickListener(new OnClickListener(){
            public void onClick(View v) {
                Intent i = new Intent(Intent.ACTION_SEND);
                i.setType("text/plain");
                String curLoc = (LocationManager)
this.act.getSystemService(Context.LOCATION_SERVICE).getLastKnownLoc
ation(LocationManager.GPS_PROVIDER).toString();
                i.putExtra("secret2", curLoc);
                this.startActivityForResult(i, 0);// SNK
            }
        });
    }

    protected void onActivityResult(int rq, int rs, Intent i) {
        …
        StringBuilder sb = new StringBuilder();
        sb.append("secret");
        sb.append("2");
        String sinkData = data.getExtras().getString(sb.toString());// SRC
        FileOutputStream outputStream;
        …
        // check perm
        if (checkCallingPermission("android.permission.WRITE_EXTE
RNAL_STORAGE")==PackageManager.PERMISSION_GRANTED) {
            outputStream.write(sinkData.getBytes());// SNK
        }
        ...
    }
}
```

(C)

```
public class Echoer extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        Button b = (Button) findViewById(R.id.b);
        b.setOnClickListener(new OnClickListener(){
            public void onClick(View v) {
                // check emul
                if (!android.os.Build.BOARD.contains("goldfish")) {
                    Intent i = getIntent();// SRC
                    this.setResult(0, i);// SNK
                }
            }
        });
    }
}
```

Fig. 1. Code snippets of the app set SWE.

The SendSMS app get device's id by calling the sensitive API *getDeviceId()*, and sends the private data to other apps for returned results using the implicit Intent call *startActivityForResult()*. Once some app receives the Intent (as long as the Intent matches its Intent Filter) and replies with exactly the received Intent by calling *setResult()*, the Echoer app for instance, the callback method *onActivityResult()* of SendSMS will be called by Android SDK. This method sends replied data outwards in SMS message by calling the sensitive API *sendTextMessage()*. Note that Android framework requires the SendSMS app declare *READ_PHONE_STATE* and *SEND_SMS* permissions to use those two sensitive APIs, while the Intent receiver app Echoer need none of such declarations. It is similar for the WriteFile app.

Neither SendSMS nor WriteFile can leak private data independently. They rely on Echoer to pass on those data to avoid merely intra-component data flows.

We further add the challenging *stateful operations* in SWE. For example, the field of the Intent that SendSMS sends out contains a key constructed by StringBuilder. This method appends "*1*" to the string "*secrete*". When SendSMS receives the echoed Intent, it only sends out the data specified by the key "*secrete1*" in SMS.

We also include *runtime conditional execution*. For instance, WriteFile checks permission declared by the caller component, and it won't write files if the caller does not have the permission to access SD card. Echoer, for another instance, won't send sensitive data via SMS when resided in an emulator, thus circumventing detection.

4 Analysis Method

The workflow of IacCE can be depicted as Fig. 2, which proceeds as follows.

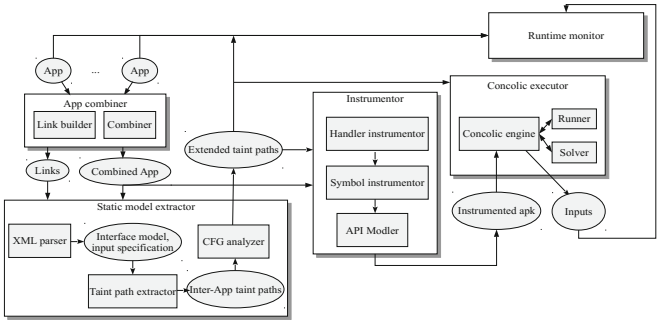

Fig. 2. Overview of IacCE.

- App combination: The apps under analysis are analyzed for IAC/ICC links and are combined as a single app.
- Static model extraction: We extract interface model, input specifications, control flow graph, and sensitive Inter-App paths. We further extend taint paths with implicit-control-flow-dependent event-chains.
- Instrumentation: The combined apk file will be instrumented for Android life-cycle entry points, event handlers, symbolic registers, register-related assignments along the extended taint paths, and user-specified external APIs.
- Concolic execution: It executes the instrumented app in emulator, performs symbolic tracing, and generate app inputs.
- Runtime monitoring: A simple dynamic monitor is implemented by running and observing the original apps with inputs generated by concolic executor.

4.1 App Combination

The first phase of IacCE builds links and combines apps in the set.

(1) ICC link exaction: An ICC link [1] is used to link two components in which the source component contains an ICC method *m* that holds explicit/implicit Intent information to access the target component *C*. Our extraction includes identifying ICC methods and Intent information, identifying target components by parsing the Intent

Filters statically declared in manifest file or dynamically defined in Java bytecode, and finally matching ICC methods with target components according to [5].
(2) App combination: In order to perform Inter-App analysis, we combine multiple Android apps to a single app in a naïve way, by extracting components and UI layout files of each app and repacking them into one apk file. This combination eases instrumentation and concolic execution of the apps, because we can consider the apps as a whole without the need of dynamically coordinating them.

4.2 Static Model Extraction

This phase produces following models.

(1) The interface model: It provides information about all input fields, as well as information about the Android IPC message (i.e., Intent) handled by Activities. All Android components contained in the app and Intent information can be decided by parsing the manifest file. Input fields can be obtained from the layout XML files.
(2) The Inter-App taint paths: Those paths cross app-boundaries before combination. They are computed by performing taint flow analysis on the connected app code. Blindly execution of all possible program paths is boring, and instructions which transmit sensitive information are better places which deserve our focus.

Before path extraction, we need to do some connection. Each ICC method call will be replaced with an instantiation of the target component with the appropriate Intent. And a dummyMain method will be generated for each component where all the life-cycle and callback methods are modeled.

After specifying sensitive source-and-sink APIs, we can then apply static taint flow analysis to find out all those Inter-App taint paths. For more details please refer to [1].

(3) The extended taint paths:

Firstly, we add supportive method calls to the paths. Taint paths only contain taint-data transmitting instructions, which may be not able to execute all by themselves. For the example shown in Fig. 3, the taint path we get is {*getDevId*, i1} => {*sendSMS*, i2}. We need additionally include callers of *getDevId()* and *sendSMS()*, that is, *onCreate()* and *onResume()* for *getDevId()*, and *onClick(b2)* for *sendSMS()*.

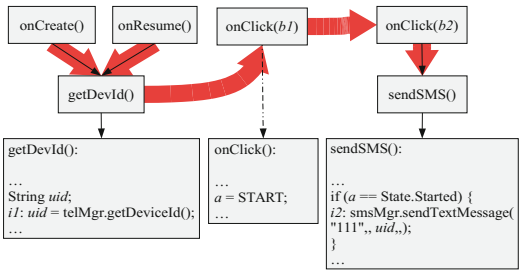

Fig. 3. Example of taint path extension. (Color figure online)

Secondly, we consider supportive event-chain extension, which is inspired by [9]. We examine implicit-control-flow dependent events with regard to the branch conditions of those extracted taint paths, for which we here only consider channels of static fields. Other channels such as file system and network should be future works. We say two events are dependent when the field read by one event is previous written by the other. For each dependency, we added a directed edge. As shown in Fig. 3, the event handler *onClick(b2)* contains a branch condition that depends on the global variable *a*. The require value *START* for *a* is set by the event handler *onClick(b1)*. Thus, the taint path will additionally include the edge {onClick(*b1*),} => {onClick(*b2*),}, where *b1* and *b2* are two distinct button instances.

The extended taint path for the example in Fig. 3 is depicted in red bold line.

4.3 Java Bytecode Instrumentation

The app is mostly executed normally, while only some variables have to be traced symbolically. To achieve this, instrumentation is needed, as illustrated by Fig. 4.

```
public class MainActivity extends Activity {
    ...
    protected void onCreate(android.os.Bundle)
    {
        Expression _sym_tmp_1 = null, $r2$sym, $z0$sym;
        MainActivity $r0 := @this: MainActivity;
        Button1Listener $r1 = new Button1Listener;
        ...
        a3targs$symargs = argpop(0, 0, 2);
        $r2$sym = models.strVar$sym;// Modeling user-specified external APIs or fields
        String $r2 = strVar;
        $r2 = getSolution_string("$X$sym_sample_vars__java_lang_String_strVar");// Injecting solutions
        $z0$sym = _contains($r2$sym, null, $r2, "pwd");// Symbolizing registers
        boolean $z0 = $r2.contains("pwd");
        _sym_tmp_1 = $z0$sym;
        if $z0 == 0 goto label1;// Symbolizing path conditions
        assume(_sym_tmp_1, 0, 0);
        ... /*  SNK API  */
    label1:
        assume(_sym_tmp_1, 0, 1);
        goto label2;
    label2:
        View $r3 = new View($r0.getApplicationContext());
        $r1.onClick($r3);// Inserting calls to event handlers
        return;
    }
}
```

Fig. 4. A sample code after instrumentation. Method summaries are simplified for reading.

(1) Inserting calls to event handlers: It is an optional heuristic to instrument component's default entry point, such as *onCreate()/onResume()* of Activity, to allowing for direct calls to event handlers thus simulating the injection of raw events. Events are distinguished by taint-path id, guaranteeing that only event handlers related to current taint path will be called. Although it is more general to inject raw events in the Android framework boundary, tracing the extra injection path require heavy instrumentation of Android system.

(2) Symbolizing registers, assignments and path conditions: This prepares for symbolic tracing, dumping path conditions, and overwriting registers with solutions at runtime. We need instrument registers, assignments and path conditions with their symbolic counterparts. However, we do not instrument all these occurrences in the app, rather than limit to those on the extended taint paths. We symbolically trace input-tainted variables within components as well as through Intent bundles.
(3) Modeling user-specified external APIs or fields: We specify user inputs such as UI text field according to the interface model we get in the static model extraction phase, to enable symbolic tracing of them. Besides, the target app might call external APIs or fields which may be hard to be symbolized. We model those user-specified APIs or fields by replacing the actual API methods with stub methods which return certain concrete values or even symbolic variable.
(4) Injecting solutions: After constraint solver find a solution, we replace each symbolic register or model w.r.t. the r-value of the original assignment with its corresponding solution. This is done by inserting method calls in the form of *getSolution_Xxx(String symVarName)*, where *"Xxx"* is a certain variable type.

4.4 Concolic Execution

We run and symbolically trace the instrumented combined app in Android emulators. Our extended-taint-path (ETP, for short) guided concolic execution iteratively does following procedures as depicted in Algorithm 1.

For each ETP, we first generate symbolic model/input configuration according to interface model which specifies user inputs. Secondly, the emulator's environment is cleaned and the instrumented combined apk is installed. Thirdly, we determine the default entry component by finding the root's containing component for each ETP. And then we start the component via am-start command. Fourthly, the solutions to inputs and modeled APIs will be injected into symbolic registers in the instrumented app*. When execution deviates from the intended taint path by branching to the wrong basic block, we dump conditions over symbolic registers in a path condition*, negate the last clause of the path condition*[1], and then feed the resulted path constraint to a SMT solver for a new solution of concrete register values leading to execution of the intended basic block. Iterate above steps until we hit the sink API for each taint path.

Note that our concolic execution is enforced only along the ETP, which avoids the notorious problem of path explosion and drives execution only along data-transmitting paths.

[1] These steps marked with the superscript "*" will be done by the instrumented app itself, rather than by the concolic engine.

Algorithm 1. ETP guided concolic execution
Input: ETP:extended taint paths, IM:interface model APK:instrumented combined apk,
Output: In:data inputs, Ev:event inputs

```
model ← getInputAndModel(IM)
i ← 0
foreach path in EPT.getPaths() do
do
  In[i] ← {}, Ev[i] ← {}
  clean()
  install(APK)
  entry ← getEntry(ETP)
  startComponent(entry)
  path ← EPT.getNextPath()
  Ev[i].add(path.getHandlers())
  In[i].add(getSolution(model))
while !isSnkHit()
i ← i+1
end
return In,Ev
```

4.5 Runtime Monitoring

Although we can just directly observe the behavior of the instrumented combined app, we should further ensure that discovered Inter-App data leaks do happen for the original apps. What is more, vendors often do not expect that data-leak issues rendered by analyst are merely related to a modified or combined version of the original app set.

```
# start entry component
adb shell am start -n app1/app1.MainActivity

# tap button1
Tap(248.0,351.0)
UserWait(4000)

# set GPS
adb -s emulator-5554 emu geo fix 121.420413 31.215345

# tap button 2
Tap(279.0,493.0)
UserWait(4000)

# send sms
adb shell am start -a android.intent.action.SENDTO -d
sms:10086 --es sms_body “secrete” --es exit_on_sent true

# press sender key to submit
adb shell input keyevent 66
```

Fig. 5. An example of Monkey scripts and am commands.

We inject events and inputs through Android Debug Bridge (adb) [11], step by step along each extended taint path, without any repacking of the original apps.

Firstly, according to the triggering order of events and inputs along the path, we generate a script for each path which contains directly injectable events. For instance, it can be a Monkey script [12] for UI events according to their location on the screen, or may be a list of the Activity Manager tool (am) [13] commands for system events according to their concrete types and the solutions to event parameters. Note that as the resolution and size of emulator's screen is fully under control, we can statically determine the location of UI widgets according to their layout files. The mixed scripts and commands shown in Fig. 5 is an example of a test that we generate.

Then, we replay UI events and system events on the original app set, by means of separate mechanisms. Raw UI events are injected directly to the emulator using the monkeyrunner tool [12]. System events are triggered using am. Specifically, we need to send an explicit Intent by am to launch the entry component of the current taint path.

Text inputs are regarded as the combination of UI events. For example, when injecting a text-input solution to an editable text widget, we generate such sequence of events as tapping the editable text widget, typing each character of the solution string by tap the corresponding soft/hard key, and typing the submitting soft/hard key.

For those modeled environment-dependent APIs or fields which cannot be directly injected, such as emulator checkers or timing bombs, we set them with the solution we get from the SMT solver facilitated by Android InstrumentationTestRunner [14].

Finally, by observing the triggered behavior for each taint paths, we can confirm the existence of data-leakage in our original app set.

5 Implementation Details

5.1 IC3 and AppCombiner

The ICC links are built by IC3 [16] and stored in database for further analysis of Inter-Component/Inter-App taint paths. ApkCombiner [17] takes all apps in the target app set as input, and outputs a combined app.

5.2 IccTA (Modified)

We use IccTA [1] to extract the Inter-App taint paths. To seamlessly integrate the static analysis process with concolic executor, we modify IccTA to store paths in the global *ArrayList* structure resided in the main entry of instrumentor. Along with that, we extract interface model and extend taint paths, thus avoiding preparing Soot [15] structures in memory for many times.

5.3 Instrumentor

We borrow ConDroid's [8] instrumentation utility, which in fact is inherited from Acteve [8]. For methods along the extended taint paths, we implement path-sensitive event-handler instrumentation, symbolic tracing of various variable types, input symbolization, and Android SDK and third-party libraries instrumentation.

5.4 Concolic Executor

We used Acteve [8] as our concolic execution engine. Extended-taint-path guided execution is already guaranteed by our customized instrumentation, while more works are involved to determine the entry component of the combined app, and to store event-and-input sequence for runtime replaying.

When encountering input-related objects which require complex instantiation, such as strings created by StringBuilder and intent data contained in Bundle, we symbolically trace them by modeling the instantiation operations of the data structure.

When those complex objects are not input-related, to avoid missing *true positive*, just add paths from intent with complex key to receiver. We are not worried about the might resulted *false positive*, as we can directly observing whether those paths actually leak data in the runtime monitor.

We integrate the string-constraint solving via z3-str SMT solver, by referring to the code of ConDroid [6] which introduces a back-tracing procedure for semantically richer solutions to registers of boolean type.

5.5 Runtime Monitor

For runtime monitor, we write a generator of Monkey scripts and am commands. It reads path information and solutions from concolic executor. We replay events and inputs contained in those scripts and commands, and observe the dynamic behavior of the whole app set.

6 Experimental Evaluation

In the following subsections, we evaluate how IacCE can be used to automatically drive sensitive data transmission, how IacCE compares with existing tools, and what capabilities IacCE has to analyze real-world apps.

6.1 Case Study: The SWE App Set

To demonstrate its capabilities in practice, we first evaluate IacCE on the SWE example described in Sect. 3. IacCE analyzes the set as follows.

- *Inter-App taint analysis*

  Two Inter-App taint paths will be extracted for the combined app. Methods containing instructions along those two paths are *SendSMS$1.onClick(View) => Echoer $1.onClick(View) => SendSMS.onActivityResult(int,int,Intent)* and *WriteFile$1. onClick(View) => Echoer$1.onClick(View) => WriteFile.onActivityResult(int,int, Intent)*.

- *Extending Inter-App taint paths*

  Supportive life-cycle handlers are added. As there exists none static-field-related event dependency, no extra extension is needed.

- *Instrumentation for event-handler calls*

Take SendSMS for example. The invocation "*ocl.onClick(view);*" is inserted at the end of *SendSMS.onCreate(Bundle)*, where *ocl* is an anonymous instance of *OnClickListener* and *view* is created according to the app context.

- *Symbolization, symbolic tracing, and solution injection*

The API/field *android.os.Build.BOARD* is path-condition related. It is modeled and to be symbolic traced during concolic execution. Every statement along taint paths will be instrumented to have its symbolic counterpart. A solution will overwrite the variable storing *BOARD*, when there is any. The instrumented code is similar to that of Fig. 4.

For each path, concolic executor iteratively performs following steps until hitting the sink APIs. Just take the first path as example:

- *Concolic execution*

We perform concrete execution of the combined app along with the symbolic tracing of the modeled *BOARD* field. As with an emulator, the branch condition in Echoer will not be taken. The path constraint *"not (Contains Xsym_android_os_Build__java_lang_String_BOARD "goldfish")*" is dumped as the sink API is not hit.

- *Solving path consraints*

The SMT solver find a solution string, say "*abc*", which is injected into the variable storing the value retrieved from the *BOARD* field.

- *Hitting the sink API*

As there is no other symbolic-variable related branch on the Inter-App taint path, the sink *sendTextMessage("10086",,msg,)* is hit in the second run of concolic execution. The iteration for the first path will then stop.

Since no symbolic-variable related branch exists for the second path, concolic execution degrades to simple concrete execution. As no relative permission granted, the branch checking the permission of Echoer in WriteFile is not taken. Thus, the sink *write(sinkData.getBytes())* will never be hit, and the leakage implied by the second taint path does not happen.

- *Replaying inputs and monitoring data-leakage*

All apps in SWE are installed in a fresh emulator. We launch the root component of the first path, i.e., SendSMS, by running an am command that sends to it an explicit starting Intent. The UI events, i.e., successive taps on the buttons, are injected via a Monkey script. The field *BOARD* is set as the solved string "*abc*". A message containing device id is observed to be sent, and we confirm the first path as Inter-App data-leaking.

6.2 Comparison with Existing Tools

Subject Apps. We choose two benchmarks, one contains **40** apps from DroidBench [22], ICC-Bench [23], and the SWE app set; the other with **77** apps is an improved version of the first one. DroidBench and ICC-Bench are frequently used as ground truth to evaluate data-leakage analyzers. DroidBench has lately added 3 app sets for evaluating IAC analyzers. As these naïve apps are initially composed for static analysis, no branches appear on any data-flow paths. Therefore, we duplicate those apps (except SWE) into two groups, and each sensitive-API call in those apps is enclosed by an additional branch condition. The resulted apps along with SWE comprise our second benchmark. All added branch conditions in one group are satisfiable, while those in the other group cannot be satisfied at runtime.

As IAC and ICC are essentially the same, we also evaluate the efficacy of existing analyzers on ICC leaks.

We compare IacCE with three existing tools: FlowDroid, IccTA, and ConDroid. We manually match the Intra-Component results of FlowDroid to report ICC leaks, just like [1] did. As with ConDroid, we add support for other components besides Activity and instrumentation of explicit intent call to enable ICC analysis, which are all described in the paper [7] while not implemented in the provided source code. We guide ConDroid with taint flows analyzed by IccTA instead of targeted call graph. COVERT is not considered as it does not perform data-leakage analysis.

For each tool on the first benchmark, each precision is 27.4%, 93.9%, 100%, and 100%, and each recall is 60.6%, 93.9%, 48.5%, and 100%.[2] The result on the second benchmark is detailedly given in Table 1.

(a) FlowDroid misses lots of ICC flows and all IAC leakages, as it cannot produce precise data-flow traces, even in the case where the components are within a single app. *(b)* ConDroid, as a dynamic methodology, reports no false positive. However, it misses a large number of leakages as it does not solve the problem of symbolically tracing implicit ICC Intents, not to mention those explicit and implicit IAC Intents. *(c)* IccTA performs well on the first benchmark of simple apps. The precision sharply degrades on the second benchmark, due to the inefficacy of static analysis for determining whether those added branch conditions on taint paths will be satisfied at runtime. *(d)* IacCE achieves higher precision and recall. It reports no false positive as it dynamically observes the execution of apps. And it does not miss any leakages as it performs conservative static taint path extraction for complicated state-full operations. **However, the 100% result does not mean IacCE always detect exactly all leaks for any app set, as will be described in Sect. 7.**

[2] Due to space limitation as well as the relative incapability of evaluating dynamic tools experienced by this benchmark, the result is not included in the paper.

Table 1. Experimental results on the second benchmark. For each app (or app set) and tool, indication of explicit or implicit ICC/IAC, true positive (TP), false positive (FP), and false negative (FN) are listed. Precision (TP/(TP + FP)) and recall (TP/(TP + FN)) are mesmerized.

App name	Explicit?	FlowDroid	IccTA	ConDroid	IacCE
DroidBench (Extended)					
startActivity1	Y	TP, FP(3)	TP, FP	TP	TP
startActivity2	Y	TP, FP(9)	TP, FP	TP	TP
startActivity3	Y	TP, FP(65)	TP, FP	TP	TP
startActivity4	N	FP(4)	–	–	–
startActivity5	N	FP(4)	–	–	–
startActivity6	Y	FP(4)	–	–	–
startActivity7	Y	FP(4)	FP(2)	–	–
startActivityForResult1	Y	TP, FP	TP, FP	TP	TP
startActivityForResult2	Y	TP, FP	TP, FP	TP	TP
startActivityForResult3	Y	TP, FP(3)	TP, FP	TP	TP
startActivityForResult4	Y	TP(2), FP(4)	TP(2), FP(2)	TP(2)	TP(2)
startService1	Y	TP, FP(3)	TP, FP	TP	TP
startService2	Y	TP, FP(3)	TP, FP	TP	TP
bindService1	Y	TP, FP(3)	TP, FP	TP	TP
bindService2	Y	FN	TP, FP	TP	TP
bindService3	Y	FN	TP, FP	TP	TP
bindService4	Y	TP, FP(3), FN	TP(2), FP(2)	TP(2)	TP(2)
sendBroadcast1	N	TP, FP(3)	TP, FP	FN	TP
insert1	N	FN	TP, FP	FN	TP
delete1	N	FN	TP, FP	FN	TP
update1	N	FN	TP, FP	FN	TP
query1	N	FN	TP, FP	FN	TP
startActivity1_src,snk	N	FN	TP, FP	FN	TP
startService1_src,snk	N	FN	TP, FP	FN	TP
sendBroadcast1_src,snk	N	FN	TP, FP	FN	TP
ICC-Bench (extended)					
Explicit1	Y	TP, FP	TP, FP	TP	TP
Implicit1	N	TP, FP	TP, FP	FN	TP
Implicit2	N	TP, FP	TP, FP	FN	TP
Implicit3	N	TP, FP	TP, FP	FN	TP
Implicit4	N	TP, FP	TP, FP	FN	TP
Implicit5	N	TP, FP(3)	TP, FP	FN	TP
Implicit6	N	TP, FP	TP, FP	FN	TP
DynRegister1	N	FN	TP, FP	FN	TP
DynRegister2	N	FN	FN	FN	TP
SWE					
SendSMS,Echoer,WriteFile	N	FN	FP, FN	FN	TP

(*continued*)

Table 1. (*continued*)

App name	Explicit?	FlowDroid	IccTA	ConDroid	IacCE
Summary					
TP		20	31	16	32
FP		126	34	0	0
FN		13	2	17	0
Precision		13.7%	47.7%	100%	100%
Recall		60.6%	93.9%	48.5%	100%

6.3 Application to Real-World Apps

Although IacCE is only a prototype by far, we successfully dynamically confirm (or eliminate false positives) several suspicious real-world leakages reported by previous static analyzers [1, 3]. Those apps are crawled from Google Play [24] and F-Droid [25].

We here describe an example of our findings. *Ermete SMS* is reported by COVERT to be vulnerable to privilege escalation if it is installed along with *Binaural beats therapy* [3]. In that case, Binaural beats therapy, designed for relaxation, creativity and many other desirable mental states and is without WRITE_SMS permission, sends an Intent with SEND action and text/plain payload data to Ermete SMS, a free web-based text messaging application that has WRITE_SMS permission.

The authors of COVERT, however, had to manually review them to confirm the vulnerability. Rather, IacCE dynamically checks the vulnerability and find it is a false positive as Ermete SMS acctually does not receive the Intent sent by the former app due to Intent field mismatching.

We further compose a malicious app which leaks location through an Intent deliberately constructed to be receivable by Ermete SMS. In this case, IacCE verified that the Inter-App data leakage does take place.

7 Discussion and Limitations

Here are some sources of unsoundness and imprecision of IacCE.

(1) Complex object symbolization. Objects which require complex initialization are difficult to symbolize and trace for symbolic execution. Although concolic execution already elevates this by concretely executing none relevant part of code and only symbolizing a rather small part, there are situations where symbolization of complex object is necessary. Presently, we tackle this problem by modeling some of the most frequently used Android complex objects.

(2) Native code, reflection and dynamic loading. Both commercial apps and malicious apps are starting to use native codes, reflection, dynamic loading, and other tricks to hide their real business logic to avoid being analyzed. This is a common issue for all existing static and dynamic analysis tools. Although researchers are trying to solve this, none satisfying solutions are available.

(3) Remote procedure calls (RPC). Besides Intent-based ICC/IAC, apps also can communicate through remote procedure calls. The latter induces method-invocation

interaction using stubs which are automatically generated by specifying component interface described in Android's Interface Definition Language (AIDL). RPC is less used than Intent. We plan to support RPC in the future.

8 Related Work

FlowDroid [4] is a state-of-the-art open-source tool for intra-component static taint analysis. It is context-, flow-, object-, and field-sensitive and Android app lifecycle-aware. However, it is confined to single components.

Didfail [2] and IccTA [1] are state-of-the-art tools for statically detecting Android ICC leaks, all based on FlowDroid. IccTA achieves better precision and recall than Didfail. It extracts the ICC links and then modifies the Jimple code of apps to directly connect the components to enable data-flow analysis between components. It then uses FlowDroid to perform high precise intra-component taint analysis and builds a complete control-flow graph of the whole Android application. IccTA allows propagating the context (e.g., the value of Intents) between Android components.

TaintDroid [29] is probably the most prominent tool for dynamic analysis of Android apps. It dynamically traces data leaks occurred during the execution of apps by applying dynamic taint analysis. Such tools are not suited for fully automated analysis since they require user interaction to drive execution of the apps.

AppIntent [6], ConDroid [7] and IntelliDroid [9], however, successfully tackles the problem of automate input generation. They use concolic execution for dynamic analysis of apps. AppIntent identifies paths which incur information leaks and performs concolic execution only on those paths. The notion of event space proposed by AppIntent is incomplete, as it only take method-call like control flow into account. ConDroid is a directed concolic analyzer for dynamic code loading in Android apps. Similar to AppIntent, it performs directed concolic execution. IntelliDroid [7] further extract event dependency according to path conditions to generate event chains.

Only quite recently, tools have emerged for IAC analysis. COVERT [3], one of such tools, detects Inter-App vulnerabilities with static model checking. It mainly performs call graph analysis for privilege escalation vulnerability rather than flow analysis for information leakage. Also, it inherits the drawbacks of static analysis.

9 Conclusion

We proposed a tool for dynamic analysis of Inter-App data leakage. It performs concolic execution guided by extended taint paths extracted by static taint analysis, and then dynamically observe and confirm whether the leakage happens at runtime. Future works include conducting tests on more real-world apps, and analyzing other types of vulnerabilities by applying model checking.

Acknowledgements. The authors would like to thank the reviewers for their detailed reviews and constructive comments, which have helped to improve the quality of this paper. This work was supported by the National Natural Science Foundation of China under Grants No. 61170286, No. 61202486.

References

1. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L., Arzt, S.: IccTA: detecting inter-component privacy leaks in android apps. In: International Conference on Software Engineering, pp. 280–291 (2015)
2. Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: Android taint flow analysis for app sets. In: International Workshop on the State of the Art in Java Program Analysis, pp. 1–6 (2014)
3. Bagheri, H., Sadeghi, A., Garcia, J., Malek, S.: Covert: compositional analysis of android inter-app permission leakage. IEEE Trans. Softw. Eng. **41**(6), 6–37 (2015)
4. Arzt, S., Rasthofer, S., Fritz, C.: FlowDroid. ACM SIGPLAN Not. **49**(6), 259–269 (2014)
5. Android documentation. http://developer.android.com/guide/components/intents-filters.html#Resolution
6. Yang, Z., Yang, M., Zhang, Y.: AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In: ACM SIGSAC Conference on Computer & Communications Security, pp. 1043–1054 (2013)
7. Schutte, J., Fedler, R., Titze, D.: ConDroid: targeted dynamic analysis of android applications. In: IEEE Conference on Advanced Information Networking and Applications, pp. 571–578 (2015)
8. Anand, S., Naik, M., Harrold, M.J.: Automated concolic testing of smartphone apps. In: International Symposium on the Foundations of Software Engineering, pp. 1–11 (2012)
9. Wong, M.Y.Y.: Targeted dynamic analysis for android malware. Dissertations & Theses Gradworks (2015)
10. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: IEEE Symposium on Security and Privacy, vol. 7, pp. 317–331 (2010)
11. Android Debug Bridge. http://developer.android.com/tools/help/adb.html
12. UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html
13. Android Activity Manager. http://developer.android.com/android/app/ActivityManager.html
14. Android instrumentationtestrunner. http://developer.android.com/reference/android/test/InstrumentationTestRunner.html
15. Soot analysis framework. http://www.sable.mcgill.ca/soot/
16. IC3. https://github.com/siis/ic3
17. ApkCombiner. https://github.com/lilicoding/ApkCombiner
18. He, J., Yang, Y.X., Qiao, Y.: Accurate classification of P2P traffic by clustering flows. China Commun. **10**(11), 42–51 (2013)
19. Zhang, Z.N., Li, D.S., Wu, K.: VMThunder: fast provisioning of large-scale virtual machine clusters. IEEE Trans. Parallel Distrib. Syst. **25**(12), 3328–3338 (2014)
20. Zhang, Z.N., Li, D.S., Wu, K.: Large-scale virtual machines provisioning in clouds: challenges and approaches. Front. Comput. Sci. **10**(1), 2–18 (2016)
21. Kirat, D., Vigna, G., Kruegel, C.: Barecloud: bare-metal analysis-based evasive malware detection. Malware Detection (2014)
22. DroidBench Benchmarks. https://github.com/secure-software-engineering/DroidBench
23. ICC-Bench. https://github.com/fgwei/ICC-Bench
24. Google play market. http://play.google.com/store/apps/
25. F-Droid. https://f-droid.org/
26. Bazaar. https://cafebazaar.ir/
27. MalGenome. http://www.malgenomeproject.org

28. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: ACM Conference on Computer and Communications Security, pp. 627–638 (2011)
29. Enck, W., Gilbert, P., Chun, B.G., Cox, L. P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: USENIX Conference on Operating Systems Design and Implementation, pp. 1–6 (2010)