Intelligent Automation And Soft Computing, 2019

Vol. 25, no. 4, 755–766
https://doi.org/10.31209/2019.100000079

Detecting Android Inter-App Data Leakage via Compositional Concolic Walking

Tianjun Wu, Yuexiang Yang

College of Computer, National University of Defense Technology, Changsha 410073, China

ABSTRACT

While many research efforts have been around auditing individual android apps, the security issues related to the interaction among multiple apps are less studied. Due to the hidden nature of Inter-App communications, few existing security tools are able to detect such related vulnerable behaviors. This paper proposes to perform overall security auditing using dynamic analysis techniques. We focus on data leakage as it is one of the most common vulnerabilities for Android applications. We present an app auditing system AppWalker, which uses concolic execution on a set of apps. We use static Inter-App taint analysis to guide the dynamic auditing procedure, so that we can target at potential Inter-App data leakage. To mitigate the exponential blow-up when auditing various combinations of apps, we introduce a novel technique called compositional concolic walking. In the end of the auditing, the event and data inputs created during concolic walking are fed to the app set. By dynamically checking the triggered data-leaking behavior, we are then able to confirm the existence of Inter-App data leakage. AppWalker takes into account both intra- and inter-app communications, and is the first research work on dynamic audit of inter-app vulnerabilities in a path-sensitive way to our knowledge. Experimental results reveal that our method can effectively detect real-world Inter-App data leakage.

KEYWORDS: Inter-App data leakage; security audit; static taint analysis; concolic walking; vulnerability analysis.

1 INTRODUCTION

THE Android mobile operation system has overtaken Windows as the most popular OS for total Internet usage [35]. It has a growing number of sources of third-party apps, i.e., app markets like F-Droid, Amazon Appstore, GetJar, et al. Thus, the security of apps in those markets becomes a large concern [23].

The inter-component communication (*ICC*, for short) model [5] of Android provides an efficient data-exchange mechanism for apps. But it also can give rise to new types of vulnerabilities such as Inter-App data leakage and collusion attacks, since a part of the functions of an app can be invoked by another app through ICC. One user is likely to have tens or even more apps installed on the device, and thus the situation of the overall security of those installed apps becomes a new interesting problem to investigate.

The challenge for auditing Inter-App vulnerabilities, e.g., Inter-App data leakage as focused on in this paper, lies in that the execution path of the vulnerable behavior distributes in multiple apps. For any app in a vulnerable app set, it can be taken as benign when being analyzed in isolation, just because it does not exhibit any sensitive behavior itself and the sensitive behavior has to be carried out as the result of the cooperation of all the apps.

While many research works have focused on auditing an individual target [4,18,19,20,26,27,28,29,30], there is only a minority of research efforts into the security audit of the entire collection of Android apps. IccTA [1] tries to find Inter-App data leakage using techniques such as static taint tracing. Convert [3] proposes to use static model checking to detect Inter-App vulnerabilities, but it cannot detect data leakage. Static analysis is essentially inaccurate and cannot avoid false positives. In this paper, instead, we propose the first dynamic audit framework for Inter-App vulnerabilities by applying concolic execution [10]. Concolic execution achieves automated input generation for dynamic analysis, by treating registers as symbols, recording path conditions w.r.t. each execution path, and then

CONTACT Tianjun Wu wutianjun08@nudt.edu.cn

solving them to get the desired inputs with a SMT solver.

Concolic execution usually faces the notorious problems of path explosion. This can be even serious for our scenario and make our analysis inefficient when analyzing a set of apps, since we have to audit different possible combination of apps. Fortunately, we find the loosely coupled and well-formed ICC interface defined by Android can be utilized to boost performance. Paths can be naturally isolated and symbolically executed according to component boundary which is unrealistic for traditional apps as class interface are diversified and ambiguous. We thus propose a new method called compositional concolic walking. Another bonus of the method is that, those path constraint segments can be re-used for a different app combination.

Evaluation over a standard benchmark as well as a real-world app set shows that our method achieves better efficiency and accuracy than state-of-the-art analysis methods for Inter-App data leakage.

The contributions of this paper are:

- Dynamic detection of Inter-App data leakage. We are the first to use concolic execution to audit Inter-App communication, to the best of our knowledge.
- Mitigation of the exponential blow-up issue for concolic execution over a set of apps. We propose a novel technique called compositional concolic walking that can efficiently determining the existence of data leakage for an app bundle.
- Implementation of our proposed method to dynamically audit a given set of apps for Inter-App data leakage. We implement our method and expect to contribute to the Android security community. We are planning to release the source code and dataset at https://github.com/leoaccount/AppWalker.

The rest of this paper is organized as follows. Section 2 introduces Android basic knowledge and several existing auditing methods. Section 3 presents the motivating example, an overview of our proposed framework, and the details of the design. The experimental results are given in Section 4. Section 5 discusses the limitations and concludes this paper.

2 BACKGROUND

2.1 Android basis

ANDROID Components. There are four categories of app components as defined by the Android framework. The Activity component displays the user interface (UI) of an app. The service component is similar to Activity, but it runs in background and mostly used for business logic which does not need display. The BroadcastReceiver component is a global receiver of Inter-Component messages. Lastly, the ContentProvider acts like a database manager for the app. Android components can be triggered by not only traditional data inputs, but also events, such as a user interaction on the Activity or a system event received by the BroadcastReceiver. The most common Inter-Component communication (*ICC*) mechanism is *Intent*. Inter-App communication (*IAC*) is similar with ICC but it stride across app boundaries.

Permission model. Android permission model involves three aspects. The API permission is used to control the high level functionality of Android framework. For example, the permission READ_PHONE_STATE which allows for reading the state of the device should be granted when the app calls any phone-state fetching APIs. The file system permission is inherited from Unix, who uses UIDs and GIDs to grant access to the storage system. The IPC permission model restricts what component can receive what Intent. This is usually defined as an Intent Filter in the Intent sender component's manifest file. Due the introduction of run-time permission model since Android 6.0 (Marshmallow), apps have to asking for permission grants at run time instead of get all permissions statically during installation. However, it may not impossible to disable permissions to pre-installed apps and the run-time permissions model is only used for apps developed using Marshmallow' SDK.

2.2 Auditing methods

STATIC auditing. The most frequently used static auditing method is static taint tracking. It starts by specifying a set of sources and sinks. It marks the source data and then pollutes every variable related to the source data in instructions that follows. Thus, this technique provides useful information of the transmission path of the data.

One of the most famous and still state-of-the-art tools that transplant static taint tracking from traditional programs to Android apps is FlowDroid [4]. Nevertheless, it is only for auditing a single app, not for a bundle of apps.

Didfail [2] uses FlowDroid to perform static taint analysis on each app component in an app. Then it synthesizes a result for Inter-Component taint transmissions by connecting the taint paths according to ICC information of the apps. Unlikely, IccTA [1] first combines all apps under test as one app, and performs taint tracking on the app like FlowDroid. The experimental results of IccTA reveal that the precision can be improved in such a way.

COVERT [3] proposes to use static model checking to auditing Inter-App vulnerabilities. It performs intra-process and inter-process control flow analysis and then uses this information collected to construct a static model for an individual app. When trying to find the vulnerability caused by IAC, it applies model checking on all models to check the existence of certain vulnerability pattern. However, since COVERT performs reachability analysis instead of taint tracking, it cannot detect data leakage.

Dynamic app auditing. TaintDroid [26] uses dynamic taint tracking to detect data flows in Android apps. It claims better accuracy than static taint tracking. But due to the inaccuracy nature of taint analysis (compared to methods like concolic execution), taint tracking is mostly used as a static preprocessor for following dynamic auditing.

Concolic execution combines concrete execution and symbolic execution to generate inputs of a program. When a concrete execution fails, it collects related path constraint and then applies symbolic execution to solve the constraint to generate a new input which can trigger the concrete execution deeper. Besides generating program inputs, concolic execution is also frequently used to verify the feasibility of program paths.

AppIntent [6] is the first to use concolic execution for Android. It generates inputs to trigger Intra-App data leakage so that security specialist can better confirm the problem. ConDroid [7] tries to detect the calling of certain malicious APIs which cannot be normally triggered in an analyzing environment, such as logic bombs, using concolic execution. IntelliDroid [9], like AppIntent, applies concolic execution for auditing data leakage within an app. The authors further provide a new method that can generate event chains to dive deep into the app.

3 OUR SOLUTION

SYSTEM Goal. We set out to build a system for effective and efficient detection of Inter-App data leakage. We have determined four design criteria that we felt such a system should satisfy so as to be useful.

1. Automated: Reduce manual efforts as much as possible.

2. *Effective*: Detect Inter-App data leakage with a small number of false positive and false negative reports.

3. *Efficient*: Speed up the analysis of a large number of apps.

4. *Robust*: Handle real-world apps correctly.

Figure 1 depicts the system architecture. In order to provide a guide to dynamic analysis, we perform a preliminary static analysis on the entire set of apps to extract Inter-App traces which may potentially cause data leakage. The apps in the bundle are assembled as one app. We use an improved version of concolic execution to determine the feasibility of each Inter-App taint trace and generate inputs which can trigger the execution of the trace. We feed the generated inputs to the apps of the app bundle in a controlled sequence. Data leakage can be confirmed if the Inter-App taint trace actually gets executed and leaks data at run time.

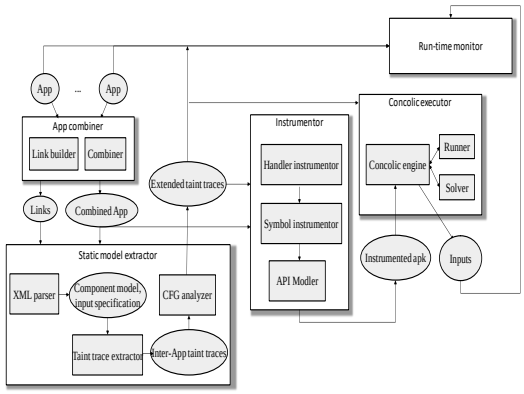

Figure. 1. System overview.

Motivating Example. We use the set of apps described in Fig. 2 for illustration, which is called the SWE app bundle. The name is the first letter of the name of each of the three apps. SendSMS and WriteFile each can potentially leak private data. But the two vulnerable behavior both need the support of the app Echoer (acts like an agent) to transmit the data. Note that Echoer itself does not leak data, because it do not access sensitive APIs at all. The SWE bundle is inspired by the example given in [2]. However, we add many challenging characteristics to prohibit existing static analysis based methods for Inter-App auditing. One is the *stateful operations*. SendSMS sends an Intent whose key happens to be dynamically created using StringBuilder. The resulted key is the concatenated string "*secrete1*". The other is *conditional execution*. Just take WriteFile as an example, it contains braches which depend on user inputs (i.e., *getData()*).

4 INTER-APP TAINT TRACE EXTRACTION

WE use static analysis to preprocess the set of apps before dynamically auditing then. A component model is firstly extracted from then entire app bundle. Based on it, we extract and then extend Inter-App taint traces to later guide our dynamic analysis.

Component model extraction. As with the static analysis phase of our method, we first collect overall information of app components for each app in the bundle. The component model (*CM*) describes the type of IPC messages sent from or can be received by a component. The manifest file of an app determines: (a) the unique package name of the app, (b) the name of all Android components within the app, and their capabilities (e.g., which Intent can handle); (c) the main process; (d) permissions required by the app (defined in *Intent Filter*); (e) permissions needed by another app to access this app's component. The properties of Intents defined in the manifest file include <action android:name/> (the action of the receiver component, e.g., making a phone call when receiving an Intent with action *ACTION_CALL*),

(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);
                if (System.currentTimeMillis()>1483228800) {
                    this.startActivityForResult(i, 0);// SINK
                }
                ...
            }
        });
    }

    protected void onActivityResult(int rq, int rs, Intent i) {
        ...
        if (android.os.Build.BRAND!=null) {
            String msg = i.getExtras().getString("secret1");// SRC
            SmsManager.getDefault().sendTextMessage("10086",,msg,,);// SINK
        }
        ...
    }
}
```

(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);
                if (getData()=="broadcast") {
                    this.startActivityForResult(i, 0);// SINK
                }
                ...
            }
        });
    }

    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) {
            if (!android.os.Build.BOARD.contains("goldfish")) {
                outputStream.write(sinkData.getBytes());// SINK
            }
        }
        ...
    }
}
```

(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);// SINK
                }
                ...
            }
        });
    }
}
```

Figure. 2. An illustrative app set (SWE).

<category android:name/> (the situation in which the action can be taken, e.g., being able to get launched in a browser with *CATEGORY_BROWSABLE*), <data android:mimeType/> (the type of the data transmitted). ICC links [1] are then extracted for linking components using Intent information. Two components can be linked only when the Intent sent form a component meets the constraints of the Intent Filter of the other component. Sometimes Intent Filters are dynamically defined in the code instead of the manifest file, for which we also need extract accordingly. This process is same with IccTA, using IC3 [16] to build links and then storing them in a database.

App assembling. We need first to assemble the apps in the bundle to enable overall static analysis. We use ApkCombiner [1] to achieve this goal. All components for each app are extracted and then packed into on single apk file. As with the manifest file of the assembled app, we just merging all manifest files from all apps. Since different apps have different default entry component, we do not make a specification. How to entry the assembled app should be determined according to what execution path we expect (see the dynamic analysis section).

Inter-App taint trace extraction. Static taint traces can be extracted by IccTA which is based on soot [15]. However, we have to make some preprocesses before that. Firstly, a call to an Android ICC method (e.g., *StartActivity(intent)*) should be replaced with a call to a function that initialized the receiver component (e.g., func(){new receiverActivity(intent)}), so that the call path is now connected. Secondly, in order to explicitly execute all components (they are originally implicitly arranged by the Android framework, which cannot get traced), a dummy main method (a new default program entry) that initializes all components of the app is added.

Now we can perform static taint analysis using a modified version of IccTA on the assembled app. The sensitive APIs we specified which may introduce sensitive data (source APIs) or leak the data (sink APIs) are the same with IccTA. By marking the sensitive data fetched by a source API and tracing instructions which transmit the sensitive data until reaching the sink API, we then have a vulnerable taint trace which may cause data leakage.

Trace extention. A taint trace consists of instructions which are only directly related to the data transmission, so it may not be executable. According to Android lifecycle state transition, for each event handling method that contains instructions on a given taint propagation path, we forward the prerequisite component that eventually starts up the activity that contains the handler to the instructions. This method is illustrated in Fig. 3, in which with regard to the tainted trace $\{getId, s1\} => \{send, s2\}$, the supporting event handlers $onCreate(), onStart(); onClick(btn2)$ are added each for the two methods on the trace.

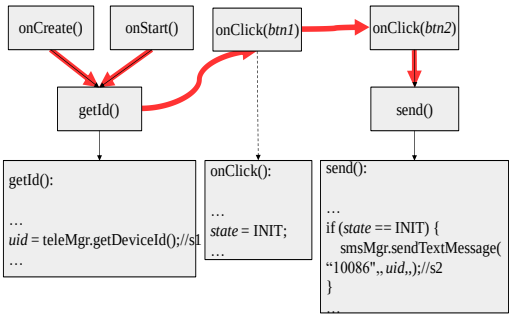

Figure. 3. An instace of extended taint trace.

Furthermore, we consider extending supportive event-chains by comprehensively examining implicit-control-flow dependencies on branch conditions. Various channels (i.e., heap variable, file system, Android framework, and environment data) are covered, instead of merely static fields. [9] only tracks static field for event dependency, which is insufficient. The dependency between two events is caused due to concurrent reading/writing a same field, which can influence the state of the program. Fig. 3 contains an instance of button-click event *onClick(btn2)* that is dependent on another instance of button-click event *onClick(btn1)*, thus we added an edge from the latter to the former. The dependency is caused in such a way that the variable *state* in a branch of *onClick(btn2)* can be modified by *onClick(btn1*). The resulted extended trace is the bold path in Fig. 3.

4.1 Compositional Concolic Walking

JAVA Bytecode Instrumentation. Instrumentation is a process to add statements to the app under test, so that execution traces of program codes can be recorded during run time. As with concolic execution, it instruments the code to follow the propagation of symbolic variables. To achieve that, we need add direct jump instruction to event handlers, generate symbolic counterparts for variables and assignment statements, and overwriting fields using the results of SMT solvers when needed. Acteve's [8] instrumentation tool provides most of the above procedures. We added the support for tracking dependent events for a taint trace, by also instrumenting the paths from the event handler (they are extracted in the static analysis phase) to the corresponding node of the trace. Thus, during run time, the execution along the path from a dependent event to the target taint trace can also be recorded for constraint solving.

Algorithm 1. ConcolicExe

Input: PATHS: program paths (default=whole program paths, i.e., *classic concolic execution*), CM: component model, APP: instrumented apk, CUR_PATH: current path (default=the path executed when no inputs are injected)
Output: RESULTS: a *global* set of the result containers, say results who have members of cst,sat,data,events, pathSignature

```
1:  model ← CM.get(APP.pkgName, PATHS)
2:  entry ←model.getEntry()
3:  While !isSinkHit() do //if not reach path end,
or path end is not sink
      // negate one of the branches of the path
      //in any priority
4:    p'←model.genNewPath(p)
5:    solve←getSolution(p') //symbolic execution
6:    if solve.sat == True do
7:      DI←append(solve.data)
        //backtrack and search for dependent events
8:      EI←append(solve.getEventHandler(p'))
9:      clean() //clean the device environment
10:     install(APP)
11:     q←startComponent(entry,DI,EI) //concrete
execution from the component entry
(ConcreteExe())
12:     ConcolicExe(PATHS,CM,APP,q)
13:   end
14:res← new result()
15:res.pathSignature ←p.signature
16:res.cst ←p
17:if isPathFeasible(p) do //sink is reached and
path constraint is satisfiable
18:  //store the path result
19:  res.sat ←True
20:  res.data ←DI
21:  res.events ←EI
22:else
23:  res.sat ←False
24:end
25:RESULTS.add(res)
26:end
```

Concolic Execution. We now apply concolic execution to judge the feasibility of the Inter-App taint traces, i.e., to judge whether they can actually execute at run-time. All apps relative to the combined app bundle are instrumented as aforementioned and now fed to a concolic executor.

Classic concolic execution [22] explores all program paths by combining concrete execution and symbolic execution. Several modifications have to be made to the original concolic execution procedure so that to make it adapted to the scenario of Android apps, similar with [6,7,8,9]. Symbolic model/input configurations are firstly generated according to the component model which specifies user inputs. As given in Algorithm 1, it then performs following procedure until a sink (some termination condition such as program exit or pre-defined data-leak point) is hit: select one *branch* condition of the *constraint* $p=b_1b_2...b_n$ (a *path/trace constraint* is the conjunction of branch conditions) of the path that have been traveled by far, say b_n; negate the selected branch and we get a new path constraint $q=b_1b_2... \neg b_n$; if q is satisfiable, concrete execution is conducted to the

corresponding program path using some solution to *q* and continue concolic execution for the new path *q*; once a sink is hit, the constraint solutions (i.e., program inputs to trigger the path) are returned. Thus, by searching recursively, concolic execution can make a traversal of all paths in a given program. The results of concolic execution are stored in the data structure RESULTS, which a global set of the result containers, say results. Each result corresponds to a path such that result.cst is the path constraint, result.sat is the satisfiability of the constraint (TRUE/FALSE), result.data/result.events is the data/event inputs needed to trigger the execution of the path, and result.pathSignature is the signature of the path.

We apply Acteve [8] as the concolic engine, integrate the z3-str SMT solver which has the solving capability of string-related constraints, and reuse a back-tracing procedure of ConDroid [6] collecting semantically richer solutions Boolean registers. For objects which are user-input-related and require complex instantiation (e.g., strings of StringBuilder and intent data stored in the Android *Bundle* data type), we model the instantiation methods of those data structures and thus can track them symbolically. If a complex object is not user-defined (i.e., nothing to do with user inputs), we add paths from intent which contains the object to the receiver component to avoid missing potential data leakage.

Concrete execution involves following steps. Firstly, the environment is cleaned for the emulator device, where the instrumented app is installed. Secondly, the default entry component is determined by looking for the component that contains the start node of each path. Thirdly, the component is started by calling an am-start command. We also need to inject the solutions (i.e., data inputs and concrete values of modeled APIs) into symbolic registers of the instrumented app.

One specialty with Android apps is that we need not only generate data inputs but also event inputs to trigger a program path. Since we have connected the discreet event space of apps, we can simply backtrack from each program statement to find event handlers which are data- or control- dependent to it. The dependency has been modeled in the taint trace extension section, and what is left now is just back tracking along that model (*getEventHandler()*, line 8 in Algorithm 1).

Fig. 4 gives an example of concolic execution. The gray and solid nodes are program terminals, dashed lines are dependent events, and the dark regions are the vulnerable paths containing data leakage. Concolic execution can firstly explore the path $D_1>0 \wedge D_2.Contains(\text{'goldfish'})$ before the vulnerable path $D_1>0 \wedge \neg D_2.Contains(\text{'goldfish'}) \wedge Sink_1$, and there can be much more redundant paths to explore if we have a more complex real-world app. When we are along the path $D_1>0 \wedge \neg D_2.Contains(\text{'goldfish'}) \wedge Sink_1$, we also backtrack each node and find two relative events E_1, E_2 which need be injected as input so as to trigger the current path. To satisfy the path condition $\neg D_2.Contains(\text{'goldfish'})$, a SMT solver is used and it generates a data input say 'abc' that does not contain the string 'goldfish'.

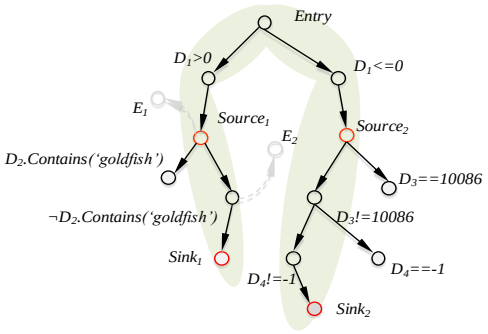

Figure. 4. A demonstration of concolic execution for Android apps.

Combinative Concolic Walking. Classic concolic execution is well-known for its inefficiency due to the path explosion problem, since there can be a large number of paths in a program. Therefore, we propose to use the previously extracted Inter-App taint traces to guide the execution, which is called concolic walking. By using the notion of "walking", we mean that the execution is performed step-by-step along the Inter-App taint traces and within the input space of data and events.

A first design that can be proposed is directly enforce the concolic walking along Inter-App taint traces for the entire app bundle that is assembled as a unique app by AppCombiner [17]. This forms the basis of our previously proposed prototype IacCE [21]. This method requires enumerating all app bundles by trying every combination of different apps, therefore we can it *combinative concolic walking*. As depicted in Algorithm 2, we first generate all possible combinations (i.e., bundles) of apps and then perform concolic walking for each entire bundle. Since we have previously connected the non-deterministic event space by inserting direct jumps between app entries, concolic execution thus can now smoothly "*walk*" through the bundle. Moreover, given the Inter-App taint traces, the concolic procedure is now forced to walk only along those traces. This is guaranteed by our Bytecode instrumentation and run-time enforced by dump branch conditions represented by symbolic registers, negating one of the clauses (i.e., branches) of a path once the path deviates from Inter-App taint traces (in Algorithm 1, line 4), and ask z3 to solve the respective constraints for the path. The solution of new values of variables will lead the execution of intended Inter-App taint traces. Once an Inter-App taint trace is considered feasible (i.e., the target API is

hit) by concolic walking, it generates and returns data and event inputs to trigger each of the traces.

Algorithm 2. Combinative Concolic Walking

Input: CM: component model, BUNDLES: all app bundles under test
Output: IDI2: Inter-App data inputs, IEI2: Inter-App event inputs

```
1: IDI2←{}, IEI2←{}
2: foreach bundle in BUNDLES do
3:   T2←getTraces(bundle)  // Inter-App taint traces
4:   reses←ConcolicExe(T2.traces, CM.get(bundle.pkgName),bundle)
5:   foreach res in reses do
6:     if res.sat == True do:
7:       IDI2.put({res.pathSignature:res.data})
8:       IEI2.put({res.pathSignature:res.events})
9:     end
10: end
11:end
```

Compositional Concolic Walking. By restricting the searching space, we save the execution cost of classic concolic execution. However, for each combination of apps, combinative concolic walking will have a fresh try of concolic walking along the respective traces, thus the cost can still be expensive. Thus we propose the second design, called *compositional concolic walking*. The method includes two phases – Intra-App constraint trawling (Algorithm 3) and compositional execution (Algorithm 4).

Algorithm 3. Intra-App Constraint trawling

Input: CM: component model, APPS: all apps under test
Output: IC1: Intra-App constraints

```
1: IC1←{}
2: foreach app in APPS do
3:   T1←getTraces(app)  //unlike for assembled app, getTraces() returns all trace segments within the app which appear in any Inter-App taint traces
4:   if T1.signature ∉ IC1.keys do
5:     res←ConcolicExe'(T1.traces, CM.get(app.pkgName), app) // do not perform the last step of concrete execution in ConcolicExe()
6:     foreach res in reses do
7:       if res.sat == True do:
8:         IC1.put({T1.signature: <res.cst,res.sat>})
9:       end
10:    end
11: end
12:end
```

Algorithm 4. Compositional execution

Input: CM: component model, BUNDLES: all app bundles under test, IC1: Intra-App constraints
Output: IDI2: Inter-App data inputs, IEI2: Inter-App event inputs

```
1: IDI2←{}, IEI2←{}
2: foreach bundle in BUNDLES do
3:   T2←getTraces(bundle)  // Inter-App taint traces
4:   foreach Inter-App trace t in T2 do
5:     foreach Intra-App trace t1 in t do
6:       <cst,sat>←IC1.get(t1.signature)
7:       if sat == False do
8:         break
9:       end
10:    end
11:    solve←getSolution(t)
12:    isFeasible←ConcreteExe(bundle,t)  //concrete execution of the Inter-App taint trace
13:    if solve.sat == True and isFeasible == True do:
14:      IDI2.put({t.signature:solve.data})
15:      IEI2.put({t.signature: solve.getEventHandler(t)})
16:    end
17: end
18:end
```

In the first phase, we trawl Intra-App constraints by performing concolic walking within each apps for a given app bundle. We use the operator ConcolicExe'() instead of ConcolicExe() to mean that we do not need the last step of concrete execution during concolic walking, as the purpose here is trawling constraints rather than executing the apps. The constraint and satisfiability of each trace is then stored for next phase. The concolic procedures are guided by Intra-App taint traces, which also appear on the Inter-App taint traces we extracted in the static model extraction section.

Then in the second phase, we set out to determine the feasibility of the Inter-App taint traces and thus determine whether data leakage happens or not. For each Inter-App taint trace in the current app bundle under test, we first check the feasibility of the Intra-App taint traces contained within that trace. Any infeasibility of Intra-App taint traces will lead to the infeasibility of the entire Inter-App taint trace, for which case we just skip and change to another Inter-App taint trace. If and only if all Intra-App taint traces are satisfiable, we call the Z3 solver to solve the constraint w.r.t. the entire Inter-App taint trace and generate inputs which can trigger the execution along that trace.

The effectiveness of our method can be illustrated by the example in Fig. 2, which corresponds to the illustrative SWE app set. We use the entity graph [3] to exhibit Inter-App data leakage. In an entity graph, every entity represents an app, each rectangle in an entity represents a component of an app, each bold swallow-tail form that embedded at the left size of a

component stands for a public ICC interface, a labeled swallow-tail form is a intent message, bold arrows are ICC channels, dashed arrows are event-dependency, and each node means a branch condition or prerequisite events of a path. There are two potential Inter-App vulnerable paths which are extracted by our previous static Inter-App taint trace extraction, $Src_1 \Rightarrow B1 \Rightarrow intent1 \Rightarrow B5 \Rightarrow intent3 \Rightarrow B2 \Rightarrow Snk_1$ (t_1) and $Src_2 \Rightarrow B3 \Rightarrow intent2 \Rightarrow B5 \Rightarrow intent3 \Rightarrow B4 \Rightarrow Snk_2$ (t_2). During the static model extraction process, we assembled the three apps into one single app.

Classic concolic execution will take all paths resulted from both side of branches of every path condition for the assembled app. Instead, combinative concolic walking performs both concrete and symbolic execution only along t_1 and t_2 for the assembled app, i.e., $ConcreteExe() \Rightarrow SMTsolve(\neg\neg B1) \Rightarrow ConcreteExe() \Rightarrow SMTsolve(B1 \wedge \neg\neg B5) \Rightarrow ConcreteExe() \Rightarrow SMTsolve(B1 \wedge B5 \wedge \neg\neg B2) \Rightarrow ConcreteExe() \Rightarrow Snk$ and $ConcreteExe() \Rightarrow SMTsolve(\neg\neg B3) \Rightarrow ConcreteExe() \Rightarrow SMTsolve(B3 \wedge \neg\neg B5) \Rightarrow ConcreteExe() \Rightarrow SMTsolve(B3 \wedge B5 \wedge \neg\neg B4) \Rightarrow infeasible$. We have $\#ConcreteExe=\#SMTsolve=3(t_1)+3(t_2)=6$.

With compositional concolic walking, according to the Intra-App constraint trawling algorithm, we first examine all Intra-App taint traces appeared on the two paths — $Src_1 \Rightarrow B1 \Rightarrow intent1$ (s_1), $B5 \Rightarrow intent3$ (s_2), $Entry \Rightarrow B2 \Rightarrow Snk_1$ (s_3), $Src_2 \Rightarrow B3 \Rightarrow intent2$ (s_4), $Entry \Rightarrow B4 \Rightarrow Snk_2$ (s_5) — using concolic walking. E.g., for $Src_1 \Rightarrow B1 \Rightarrow intent1$, we have $ConcreteExe() \Rightarrow SMTsolve(\neg\neg B1) \Rightarrow ConcreteExe()$. Then we apply one more symbolic execution for each of the two Inter-App taint traces t_1, t_2; as $B4 \Rightarrow Snk_2$ can be found infeasible (Echoer does not have the permission to call the WriteFile Activity) in advance during Intra-App constraint trawling, the execution on t_2 is avoided. In all, for t_1 we have $\#ConcreteExe=\#SMTsolve=1(s_1)+1(s_2)+1(s_3)+1(t_1)=4$. Since the constraints can be reused, the cost of t_2 is reduced to $\#ConcreteExe=\#SMTsolve=0$. Note that, once all constraints for every apps have been trawled (the number of apps are relatively much smaller than the number of all possible combination of apps, so we will soon encounter all apps after trying several combination of apps and the trawling can end in a reasonable small time), the execution cost for any Inter-App taint trace can be reduced to $\#ConcreteExe=0,\#SMTsolve=0$ or 1. That is, given pre-computed Intra-App constraints of s_i (i=1,2,3,4,5), the total cost becomes merely $\#ConcreteExe=0,\#SMTsolve=1$ for compositional concolic walking of t_1, t_2 in the example in Fig. 2.

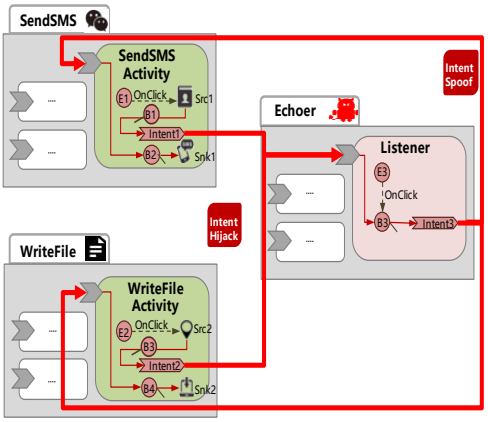

Figure. 5. Compositional concolic walking for the illustrating app bundle.

Given an arbitrary set of apps under test, let us analyze the cost of time and space for the above algorithms. Let n be the number of apps in the bundle under test, b_i be the branch count of the i^{th} app, and t_i be the branch count of the Intra-App taint trace of the i^{th} app. And let Δ_s, Δ_r each be the maximal time needed to solve a path constraint and to initiate and run an app in the device. The time complexity of concolic execution depends on the number of paths. Thus, we have $O((\Delta_s+\Delta_r)\cdot 2^{\sum_{i=1}^{n} b_i})$ and $O((\Delta_s+\Delta_r)\cdot 2^{\sum_{i=1}^{n} t_i})$ for classic concolic execution and combinative concolic walking respectively. The space complexity is the same with time complexity, since concolic execution stores each path during the recursive search procedure. We can see from the analysis that, by forcing execution along taint traces, the latter two algorithms avoid path enumeration and are thus much more efficient.

As with compositional concolic walking, the cost becomes $O((\Delta_s+\Delta_r)\cdot\sum_{i=1}^{n} 2^{t_i} + \Delta_s \cdot 2^{\mu\cdot\sum_{i=1}^{n} t_i}) = O(\Delta_r \cdot \sum_{i=1}^{n} 2^{t_i} + \Delta_s \cdot (\sum_{i=1}^{n} 2^{t_i} + 2^{\mu\cdot\sum_{i=1}^{n} t_i}))$, where μ is the fraction of Inter-App taint traces get solved in line 11 of the compositional execution algorithm, $O((\Delta_s+\Delta_r)\cdot\sum_{i=1}^{n} 2^{t_i})$ is the cost of Intra-App Constraint trawling, and $O(\Delta_s \cdot 2^{\mu\cdot\sum_{i=1}^{n} t_i})$ is the cost of compositional execution. This cost is much lower than combinative concolic walking in two reasons. Firstly and most importantly, we manage to avoid most executions of apps by reusing the concolic execution results of Intra-App taint traces once another bundle of

apps contains the same traces. The time for running an app in mobile devices or device emulators (in seconds) is much slower than that that for solving path constraints and executing the analysis algorithms which are implemented as PC programs (in milliseconds per instruction). We have now moved the overhead Δ_r to Intra-App Constraint trawling, and only leave the overhead Δ_s to compositional execution. The reuse of trances also reduces the time complexity of concolic execution procedure itself. Secondly, due to the fact that static analysis incurs a large number of falsely reported (i.e., infeasible) data leakages, only a small portion μ of Intra-App taint traces can be pruned before the final execution of the entire Inter-App taint trace.

4.2 Controlled Execution & Check

Finally, events and inputs generated by concolic execution are injected through adb, am commands, and InstrumentationTestRunner, step by step along each path in the event-dependency graph. By observing the vulnerable-path related behavior, an analyst manages to confirm the existence of Inter-App vulnerable API calls in the original app set.

This step is necessary, as an analyst can fully confirm the data leak the analyzer reports only if the vulnerability does happen under controlled inputs. Besides, since showing how to reproduce the vulnerability is the most critical part of a vulnerability report, by directly triggering the data leakage we can make it more comprehensible and convincible to app vendors. Also note that we should never base our report on modified apps (e.g., the assembled app for the app bundle), we have to control the execution and cooperation of the set of original apps by only manipulating data and event program inputs.

Here are some detailed designs of this module of AppWalker. For each Inter-App taint trace, we generate and then inject events/data inputs sequentially along the trace. UI events are generated by Monkey script [12] and injected by monkeyrunner tool [12], whose positions on the device screen can be statically determined by referring to the layout configurations of apps. System events are generated and injected via the Activity Manager (am) commands [13]. Data inputs, which have been previously generated by concolic execution, are now directly fed [14] to the apps. When starting the entry component of the taint trace, we send a start Intent to it, using Android Debug Bridge (adb) [11]. After being properly injected all inputs, the set of apps are now triggered, and we just need to observe whether the Inter-App data leakage is replayed to confirm the vulnerability.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setting

WE design the experiments by evaluating our design criteria. The effectiveness of our method is demonstrated on a standard benchmark, which makes it easy for comparing various approaches. The efficiency and robustness is evaluated on a larger set of real-world apps. Our experiments are performed on a laptop (Duo T5550, 2 cores, 4G RAM). AppWalker is written in Java and apps run in Android emulators.

Benchmarks:

- IAC-Bench. It is a mixture of DroidBench [4], ICC-Bench [1], and SWE. DroidBench and ICC-Bench are standard Android application benchmarks used as the ground truth of the effectiveness of both Intra-App and Inter-App data-leakage detectors. We have both ICC and IAC data leak instances, due to the lack of IAC benchmarks and the fact that ICC and IAC are essentially similar.
- IAC-Bench-ext. It has 77 apps which are derived from IAC-Bench. DroidBench and ICC-bench are initially proposed for testing static tools, few run-time constraints exist. In practical, malicious app authors usually add such constraints to make traditional static analysis fail to detect vulnerable app behaviors. To better evaluate dynamic methods and to compare them with static methods, we add branch conditions (collected from some popular malware benchmarks) to protect each sensitive API for the 28 apps in DroidBench, and 9 apps in ICC-Bench. Since there can be two possible branches for each branch condition, we have two set of apps for the IAC-Bench-ext benchmark — one set has been added branch conditions which are all satisfiable (IAC-Bench-ext-sat), the other set has branch conditions all unsatisfiable (IAC-Bench-ext-unsat).
- F-Droid. F-Droid [25] is a free application market for Android users. As most apps in the F-Droid market are popular real-world apps appearing in Google Play [24], many research works have used it as a standard benchmark. All apps in the site are required to open-sourced, which enable us to manually dive deep into the code to confirm the vulnerabilities we found. We randomly selected 100 apps in the F-Droid app collection provided by [3].

Methods compared:

- FlowDroid. As the same trick with [1], we still can use FlowDroid to statically detect Inter-App data leakage, though it was initially proposed only for analyzing Intra-Component taint paths.
- IccTA. It performs pure static taint analysis for Inter-App data leakage, as has been introduced in previous sections. COVERT is not open sourced and cannot detect data leakage, so we do not use it for comparison.
- ConDroid. To enable the detection of data leaks rather than API invocations, we use a procedure of static Intra-App taint traces extraction to guide the

concolic execution, similar to AppWalker, instead of the original call graph guided execution. Event-dependency extraction is also added. We also modified the system to support the Android component of BroadcastReceiver and Service, besides the Activity component.

- AppWalker. AppWalker is our method of dynamic detection of Inter-App data leaks. It can also be directly used to perform Intra-App analysis.

5.2 Comparison of Various Approaches

table 1 and Table 2 give the results of different methods on the benchmarks IAC-Bench-ext-sat and IAC-Bench-ext-unsat.

For FlowDroid, IccTA, ConDroid and AppWalker, they each achieve the precision of 27.4%/13.7%, 93.9%/47.7%, 100%/100%, 100%/100% on IAC-Bench-ext-sat/IAC-Bench-ext-unsat, and the recall rate of 60.6%/60.6%, 93.9%/93.9%, 48.5%/48.5%, 100%/100%.

We can see that FlowDroid works poorly. Taint paths are isolately extracted for individual app component and then combined, so the tainted data can be mismatched. A lot of false reports are therefore generated by FlowDroid.

IccTA manages to detect the majority of leaks, but it still does not work well with dynamic program constraints. The detection rate sharply drops from 93.9% (Bench-ext-sat) to 47.7% (IAC-Bench-ext-unsat), because IccTA cannot tell whether the sensitive APIs can actually be called at run time.

The data leaks detected by ConDroid are all true positive reports, that is, it does not give false positive reports. This owns to that ConDroid is a dynamic method, among the above two methods, which is effective at pruning false positive reports. However, ConDroid missed some Intra-App leaks as it does not track data flows through implicit ICC. It also missed all Inter-App leaks since it only can analyze a single app instead of a set of apps as a whole.

Table 1 Experimental results on IAC-Bench-ext-sat. True positive: ▲ false positive: ▼ true negative: △ false negative: ▽ precision: ●=▲/(▲+▼) recall:○=▲/(▲+▽)

package	FlowDroid	IccTA	ConDroid	AppWalker
DroidBench-ext-sat				
startActivity1	▲▼	▲	▲	▲
startActivity2	▲▼▼▼	▲	▲	▲
startActivity3	▲▼(32)	▲	▲	▲
startActivity4	▼▼	-	-	-
startActivity5	▼▼	-	-	-
startActivity6	▼▼	-	-	-
startActivity7	▼▼	▼	-	-
startActivityForR 1	▲	▲	▲	▲
startActivityForR 2	▲	▲	▲	▲
startActivityForR 3	▲▼	▲	▲	▲
startActivityForR 4	▲▲▼	▲▲	▲▲	▲▲
startService1	▲▼	▲	▲	▲
startService2	▲▼	▲	▲	▲
bindService1	▲▼	▲	▲	▲
bindService2	▽	▲	▲	▲
bindService3	▽	▲	▲	▲
bindService4	▲▼▽	▲▲	▲▲	▲▲
sendBroadcast1	▲▼	▲	▽	▲
insert1	▽	▲	▽	▲
delete1	▽	▲	▽	▲
update1	▽	▲	▽	▲
query1	▽	▲	▽	▲
startActivity1 set	▽	▲	▽	▲
startService1 set	▽	▲	▽	▲
sendBroadcast1 se	▽	▲	▽	▲
ICC-Bench-ext-sat				
Explicit1	▲	▲	▲	▲
Implicit1	▲	▲	▽	▲
Implicit2	▲	▲	▽	▲
Implicit3	▲	▲	▽	▲
Implicit4	▲	▲	▽	▲
Implicit5	▲▼	▲	▽	▲
Implicit6	▲	▲	▽	▲
DynRegister1	▽	▲	▽	▲
DynRegister2	▽	▽	▽	▲
The SWE bundle				
WE set	▽	▼▽	▽	▲
Total				
▲	20	31	16	32
▼	53	2	0	0
▽	13	2	17	0
●	27.4%	93.9%	100%	100%
○	60.1%	93.9%	48.5%	100%

Table 2 Experimental results on IAC-Bench-ext-unsat. The table is organized similarly to Table 1.

package	FlowDroid	IccTA	ConDroid	AppWalker
DroidBench-ext-unsat				
startActivity1	▲▼▼▼	▲▼	▲	▲
startActivity2	▲▼(9)	▲▼	▲	▲
startActivity3	▲▼(65)	▲▼	▲	▲
startActivity4	▼▼▼▼	-	-	-
startActivity5	▼▼▼▼	-	-	-
startActivity6	▼▼▼▼	-	-	-
startActivity7	▼▼▼▼	▼▼	-	-
startActivityForR es1	▲▼	▲▼	▲	▲
startActivityForR es2	▲▼	▲▼	▲	▲
startActivityForR es3	▲▼▼▼	▲▼	▲	▲

startActivityForResult4	▲▲▼(4)	▲▲▼	▲▲	▲▲
startService1	▲▼▼▼	▲▼	▲	▲
startService2	▲▼▼▼	▲▼	▲	▲
bindService1	▲▼▼▼	▲▼	▲	▲
bindService2	▽	▲▼	▲	▲
bindService3	▽	▲▼	▲	▲
bindService4	▲▼▼▼▽	▲▲▼▼	▲▲	▲▲
sendBroadcast1	▲▼▼▼	▲▼	▽	▲
insert1	▽	▲▼	▽	▲
delete1	▽	▲▼	▽	▲
update1	▽	▲▼	▽	▲
query1	▽	▲▼	▽	▲
startActivity1 set	▽	▲▼	▽	▲
startService1 set	▽	▲▼	▽	▲
sendBroadcast1 set	▽	▲▼	▽	▲
ICC-Bench-ext-unsat				
Explicit1	▲▼	▲▼	▲	▲
Implicit1	▲▼	▲▼	▽	▲
Implicit2	▲▼	▲▼	▽	▲
Implicit3	▲▼	▲▼	▽	▲
Implicit4	▲▼	▲▼	▽	▲
Implicit5	▲▼▼▼	▲▼	▽	▲
Implicit6	▲▼	▲▼	▽	▲
DynRegister1	▽	▲▼	▽	▲
DynRegister2	▽	▽	▽	▲
The SWE bundle				
SWE set	▽	▼▽	▽	▲
Total				
▲	20	31	16	32
▼	126	34	0	0
▽	13	2	17	0
	13.7%	47.7%	100%	100%
	60.6%	93.9%	48.5%	100%

Our method AppWalker achieves the best results for both benchmarks, i.e., the highest precision and recall rate. We are able to detect more leaks than IccTA, because we have modified IccTA by conservatively adding paths from an intent (when the intent contains complex object) sender to the receiver to avoid missing potential leakage. False positiveness, if there exists any, can be easily pruned during dynamic analysis. Most importantly, when analyzing Inter-App information leaks for a set of apps, we neither miss any data leaks nor introduce false positive reports.

5.3 Application to Real-world Apps

In this section, we evaluate AppWalker on the selected real-world apps from F-Droid. The overall result is given in Table 3.

Table 3 #Apps from 100 F-Droid apps which are found to be vulnerable. The result is represented as "#Statically detected / #Dynamically confirmed by the detector itself / #Actual data leaks"

	Intra-App	Inter-App
AppWalker	31/3/3	53/5/5

We here provide an interesting vulnerable case involving 3 apps. One app called Ermete SMS, which exposes its ICC interface and has the WRITE_SMS permission, is previously reported by COVERT to be exploitable when another app, Binaural beats therapy, runs in the same device simultaneously. The latter app, which does not have the WRITE_SMS permission, can escalate its privilege by crafting an intent and send it to the former app. This vulnerability, however, is a false positive report as the field of the intent sent from Binaural beats therapy mismatches with that of Ermete SMS and thus the latter can never receive the ICC message. Instead, we find another app, Hesabdar, whose TransactionsActivity component handles user money transaction and sends the account information as payload of an implicit intent to another component. However, there is no guarantee for this transmission as it can easily be intercepted. To demonstrate that, we further compose an exploiting app which first hijacks the intents containing the account information from Hesabdar, and then sends a spoofed intent to Ermete SMS filled with account information and adversary phone number as the payload.

6 CONCLUSION

WE propose a method for dynamic auditing a set of apps for Inter-App data leakage, which is caused due to the cooperation of multiple apps. Our approach consists two phases: firstly statically extract potentially vulnerable Inter-App traces and then dynamically analyzing those traces using concolic execution for confirmation. Several techniques are proposed to increase the accuracy and boost the speed of analysis.

7 ACKNOWLEDGMENTS

THIS research was supported by the Mobile Research Funding of Chinese Education Ministry (MCM) under Grant No. MCM20170404.

8 REFERENCES

Anand, S., Naik, M., Harrold, M. J., & Yang, H. (2012, November). Automated concolic testing of smartphone apps. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (p. 59). ACM.

Android Debug Bridge | Android API. http://www.android-doc.com/tools/help/adb.html.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., & McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices, 49(6), 259-269.

Bagheri, H., Sadeghi, A., Jabbarvand, R., & Malek, S. (2016, June). Practical, formal synthesis and automatic enforcement of security policies for android. In Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on (pp. 514-525). IEEE.

Ball, S., & Tuker, M. (2017). A fuzzy multi-criteria decision analysis approach for the evaluation of the network service providers in turkey. Intelligent Automation & Soft Computing, 1-7.

Build instrumented unit tests. https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.

Burket, J., Flynn, L., Klieber, W., Lim, J., Shen, W., & Snavely, W. (2015). Making didfail succeed: Enhancing the cert static taint analyzer for android app sets (No. CMU/SEI-2015-TR-001). CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH US.

Burnim, J., & Sen, K. (2008, September). Heuristics for scalable dynamic test generation. In Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering (pp. 443-446). IEEE Computer Society.

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., & Engler, D. R. (2008). EXE: automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC), 12(2), 10.

Documentation for app developers. https://developer.android.google.cn/docs/.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B. G., Cox, L. P.,... & Sheth, A. N. (2014). TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems, 32(2), 5.

F-Droid - Free and open source Android app repository. https://f-droid.org/.

Google play store. https://play.google.com/.

Li, L., Bartel, A., Bissyandé, T. F., Klein, J., & Le Traon, Y. (2015, May). Apkcombiner: Combining multiple android apps to support inter-app analysis. In IFIP International Information Security Conference (pp. 513-527). Springer, Cham.

Li, L., Bartel, A., Klein, J., & Le, T. (2014) Detecting privacy leaks in Android Apps. CEUR Workshop Proceedings (pp. 1298).

Li, L., Bartel, A., Klein, J., Traon, Y. L., Arzt, S., Rasthofer, S.,... & Mcdaniel, P. (2014). I know what leaked in your pocket: uncovering privacy leaks on Android apps with static taint analysis. arXiv preprint arXiv:1404.7431.

monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/.

Protalinski, E. (2016, October 12). Android malware numbers explode to 25,000 in June 2012. Retrieved from www.zdnet.com.

Quiroz, J. C., Banerjee, A., Dascalu, S. M., & Lau, S. L.. (2017). Feature selection for activity recognition from smartphone accelerometer data. Intelligent Automation & Soft Computing, 1-9.

Schütte, J., Fedler, R., & Titze, D. (2015, March). Condroid: Targeted dynamic analysis of android applications. In Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on (pp. 571-578). IEEE.

Using activity manager (am). http://androiddoc.qiniudn.com/tools/help/shell.html#am.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., & Sundaresan, V. (2010, November). Soot: A Java bytecode optimization framework. In CASCON First Decade High Impact Papers (pp. 214-224). IBM Corp.

Wang, P., Lu, K., Li, G., & Zhou, X. DFTracker: detecting double-fetch bugs by multi-taint parallel tracking. Frontiers of Computer Science, 1-17.

Wong, M. Y., & Lie, D. (2016, February). IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In NDSS (Vol. 16, pp. 21-24).

Wu, T., & Yang, Y. (2016, October). IacCE: Extended Taint Path Guided Dynamic Analysis of Android Inter-App Data Leakage. In International Conference on Security and Privacy in Communication Systems (pp. 317-333). Springer, Cham.

Wu, Z., Lu, K., Wang, X., & Zhou, X. (2015). Collaborative technique for concurrency bug detection. International Journal of Parallel Programming, 43(2), 260-285.

Wu, Z., Lu, K., Wang, X., Zhou, X., & Chen, C. (2015). Detecting harmful data races through parallel verification. The Journal of Supercomputing, 71(8), 2922-2943.

Yiğit Kültür, & Mehmet Ufuk Çağlayan. (2015). A novel cardholder behavior model for detecting credit card fraud. International Conference on Application of Information & Communication Technologies. IEEE.

Zareapoor, M., & Yang, J.. (2017). A novel strategy for mining highly imbalanced data in credit card transactions. Intelligent Automation & Soft Computing, 1-7.

Zhang, Y., Yang, M., Yang, Z., Gu, G., Ning, P., & Zang, B. (2014). Permission use analysis for vetting undesirable behaviors in android apps. IEEE transactions on information forensics and security, 9(11), 1828-1842.

9 NOTES ON CONTRIBUTORS

Tianjun Wu received his M.S. degrees in computer science from College of Computer, National University of Defense Technology (NUDT), China in 2015. He is currently a Ph.D. candidate in College of Computer of NUDT. His research interests include vulnerability mining, network security, and information security.

Yuexiang Yang received his Ph.D. degree in computer science from College of Computer of NUDT, China in 2006. He is currently a professor with College of Computer of NUDT. His research interests include information security, network security, architecture design of the Internet, and web service.
Email: yyx@nudt.edu.cn