2018 15th International ISC (Iranian Society of Cryptology) Conference on Information Security and Cryptology (ISCISC)
August 28-29, 2018; Shahid Rajaee Teacher Training University - Tehran, Iran

Detecting Privacy Leaks in Android Apps using Inter-Component Information Flow Control Analysis

Zohreh Bohluli
Department of Computer Eng. and IT
Amirkabir University of Technology
Tehran, Iran
z.bohluli@aut.ac.ir

Hamid Reza Shahriari
Department of Computer Eng. and IT
Amirkabir University of Technology
Tehran, Iran
shahriari@aut.ac.ir

***Abstract*—Nowadays, smartphones are ubiquitous sources of private and confidential information. Among smartphones operating systems, Android has become the most popular one in recent years. Android applications have access to different information which stored on the device so, may lead to information leaks accidentally or maliciously. Leakages stem from explicit or implicit information flows between information sources and sinks. Finding explicit flows is fairly simple whereas, implicit flows utilize more complicated structures and are more difficult to discover, as a result. Most existing tools ignore implicit flows or only consider special structures that are similar to explicit form in nature such as *if* and *switch* structures. In this paper we propose IIFDroid, inter-component information flow control static analysis tool which aims to detect information leaks generated by explicit and various forms of implicit flows within an Android application. Furthermore, we present test cases in order to examine the effectiveness of IIFDroid against implicit flows caused by more sophisticated structures like throw, polymorphism and exception-prone instructions. The experimental results on DroidBench and the developed test cases show that IIFDroid outperforms existing tools IccTA and JoDroid with 94.8% precision and 96.4% recall.**

Keywords—Android Applications, Privacy Leaks, Information Flow Control, Static Analysis, Inter-Component Communication

I. INTRODUCTION

Today, Android applications play a decisive role in our life. On average, each Android user installs 95 applications on her phone and uses 35 of them on a daily basis [1]. These apps have access to various sensitive information stored on the device such as contacts, gallery, IMEI and may send them to unwanted destinations without user consent or store insecurely on the device. Private information leakage is one of the mobile top ten risks reported by OWASP in 2016 [2]. Hence, some approaches are required to analyze application's behavior more accurately.

Based on the definition provided in [3], "privacy leaks are paths from sensitive data, called sources, to statements sending the data outside the application or device, called sinks".

Leakages may happen inside a component or between different components from one/different app(s). Privacy leaks are due to explicit or implicit information flows between sources and sinks [4]. Tracking explicit flows is much simpler whereas, implicit flows are more sophisticated and are neglected by most existing tools.

In this paper, we propose IIFDroid[1], a precise Inter-component Information Flow control static analysis tool for Android applications aiming at discovering privacy leaks within an app especially those that caused by implicit flows.

At first, IIFDroid employs precise static analysis techniques to extract System Dependence Graph (SDG) of an Android app representing explicit and implicit information flows between program statements. Then, the backward slices of sinks are computed to check whether information with a higher security level reach sinks with lower security levels or not. If so, a potential privacy leak will be reported.

The rest of the paper is organized as follows, in Section II a brief background about Android app architecture, static analysis challenges and various kinds of information flows are given. Section III reviews existing work. In Section IV, IIFDroid workflow is given then information flow control analysis and IIFDroid architecture are explained. Section V includes implementation details, Section VI shows evaluation results and Section VII concludes the paper.

II. BACKGROUND

A. Android Application Architecture

Android apps adhere to a component-based architecture [5]. There are four kinds of components: activity, service, content provider and broadcast receiver.

Activities construct user interface of an app. Services do not have user interface and are used to perform time-consuming tasks in the background. Content providers act analogous to a database and provide access to a constructed set of data. Broadcast receivers listen to global events e.g. battery is low.

[1] Online available at: github.com/zohreb71/ATLAS-IFFDroid

978-1-5386-7582-3/18/$31.00 ©2018 IEEE

Components in an app or across different apps, utilize so-called inter-component communication (ICC) methods to communicate and exchange data. These methods take a message named Intent or URI[2] (in case of content providers) as parameter. To handle ICC interactions, the target component and transferred data need to be resolved.

Finally, every app consists a metadata file named AndroidManifest.xml that keeps essential information like permissions, components and action strings that each component can process.

B. Static Analysis Challenges

Although there are different ways for Android app development, IIFDroid only concentrates on those which are developed using standard Google API in Java programming language.

Despite these apps are written in Java, they compile into dalvik bytecode. Retargeting tools like Dare [6] convert them into java bytecode and give as input to existing java static analysis frameworks. Nevertheless, there are fundamental differences between Java and Android that make precise modeling of Android app's runtime behavior more challenging.

Apps contain different components with a distinct lifecycle [4]. They do not have a main method rather Android runtime invokes various callback methods within the app based on system events to start, pause, resume and shutdown the app.

Furthermore, static analysis tools must regard system and UI callbacks. Sometimes, UI contains sensitive information sources like password fields. API calls which return their content are not in the program code [7]. So, precise modeling of metadata and layout xml files is also required.

To be precise, static analysis techniques must support analysis sensitivities [8]. Flow sensitivity is the most common one which takes the order of program statements into account. Object sensitive and field sensitive static analysis model each instance object of a class and each field of an object individually. Eventually, in context sensitive static analysis, each method call is modeled independently.

C. Information Flow Types

As mentioned earlier, leakages are due to information flows between information sources and sinks which Android app has access to. There are two types of information flows: explicit and implicit. Explicit flows caused by data dependency i.e. def-use relationship between program variables. On the contrary side, implicit flows are the result of control dependency between program statements [4]. For instance, in Fig. 1-a, line 2 defines variable x which is used in line 3. So, there is a data dependence between them, whereas, in Fig. 1-b, x controls the execution of assignment in line 5 and there is an implicit flow from x to y.

You et al. [9] studied Android bytecode comprehensively to identify all possible forms of implicit flows. They applied control-transfer-oriented semantic analysis on Android bytecode and found 54 dalvik instructions that can induce implicit flow. Finally, all founded instructions were categorized into five classes based on their underlying structure: *if-based*, *switch-based*, *throw-based*, *exception-prone-based* and *polymorphism-based*. They presented proof-of-concepts for each identified category to examine their exploitability. An overview of proof-of-concepts is shown in Fig. 2, each one consists of a 1-1 mapping between private and public data.

```
1.  void explicitFlow() {
2.       int x = source ;
3.       int y = 2 * x ;
4.       sink (y) ;
5.       }
```
(a)

```
1.  void implicitFlow() {
2.       int x = source ;
3.       if (x < max)
4.       {
5.            int y = 2 * x ;
6.            sink (y) ;
7.       }
8.  }
```
(b)

Fig. 1. Implicit vs. explicit flow

```
1.  if (high=='0') low = 0 ;
2.  else if (high=='1') low = 1 ;
```
(a)

```
1.  for (low = '0'; low <= '9' ; low++) {
2.   try { int tmp = 1 / (high- low); }
3.   catch (Exception e) { break ; }
4.  }
```
(b)

```
1.  switch (high) {
2.   case '0': low = '0' ; break ;
3.   case '1': low = '1' ; break ;
4.  }
```
(c)

```
1.  Class Poly_0 extends Poly {
2.  char f() {return '0';}
3.  …
4.  Poly poly = polys [high - '0'];
5.  Low = poly.f() ;
```
(d)

```
1.  Exception except = excepts[high-'0'];
2.  try { throw except ; }
3.  catch (Exception_0 e) {low = '0' ; }
4.  catch (Exception_1 e) {low = '1' ; }
```
(e)

Fig. 2. Implicit information flow forms in Android bytecode [9]

III. RELATED WORK

Although privacy leak is the most covered vulnerability among Android static analysis tools, only few studies have noticed implicit flows [4].

FlowDroid [7] models component's lifecycle by creating a dummy main method and performs context, flow, field and object-sensitive taint analysis to discover leakages within a component precisely. Later, authors enhanced taint analysis technique to detect leaks created by simple implicit flow structures like if and switch. FlowDroid does not model ICC communications and overapproximated inter-component communications by taking each transferred data element as a taint.

IccTA [3] adds a preprocessing step to FlowDroid in order to discover inter-component leaks. It uses IC3 [10] to build ICC links between different components. Then, it instruments the app to connect components directly and builds an Inter-procedural Control Flow Graph (ICFG) of the whole Android app. At the end, it utilizes an enhanced version of FlowDroid to perform inter-component taint analysis. So, it is similar to FlowDroid in case of implicit flows.

Joana [11] is a static information flow control analysis tool for Java programs that aims to discover all security leaks caused by explicit or various forms of implicit flows. Later in 2013 JoDroid [12], an Android front-end for Joana has been proposed to discover all types of information leaks in Android apps. Despite its goal, JoDroid runs into many troubles in practice. It not only does not handle Android-specific challenges such as precise lifecycle modeling but

[2] Uniform Resource Identifier

also it is unable to recognize information sources and sinks within an app accurately.

IV. OUR APPROACH

A. Proposed Method

The purpose of IIFDroid is discovering information leaks caused by explicit and five-fold forms of implicit flows in one component or between different components within an Android app. To achieve this goal, IIFDroid adopts inter-component information flow control static analysis.

The general workflow of IIFDroid is depicted in Fig. 3. IIFDroid takes an Android app as input and outputs all founded potential information leaks. It begins with converting the app to an intermediate representation which is suitable for future analysis. Then, it generates SDG of the program (step 2) and annotates sources and sinks with appropriate security levels (step 3).

Fig. 3. General overview of IIFDroid process

Finally, information flow control analysis is performed in step 4. Backward slice of sinks is calculated to check whether it contains an information source with a higher or non-comparable security level or not. Consequently, all founded leaks are reported to the security analyst.

B. Information Flow Control Analysis

Information flow control (IFC) [11] is a known program analysis technique for discovering security leaks in software. Language-based IFC analysis aims to establish noninterfernce. It takes program code (in source code or bytecode) as input and checks whether it obeys confidentiality and/or integrity.

Our IFC analysis technique uses SDG as the fundamental structure. SDG is a standard data structure for modeling information flows through a program. It includes Program Dependency Graph (PDG) for each procedure of program connected by the call, return and parameter passing edges.

SDG Nodes are program statements and predicates connected by two types of edges: data-dependence which models def-use relationship between variables, control-dependence that represents control dependency between nodes. We will explain IFC analysis technique in PDG. All notions are the same as SDG. However, function call and return relationship must be handled properly in SDG and only so-called realizable paths should be taken into consideration. Interested readers can refer to [11], [13] for more information about SDG and slicing.

In a PDG G = (N, →), backward slice[3] of x consists all nodes possibly influencing x and is computed as:

$$\boldsymbol{BS(x) = \{y \mid y \rightarrow^{*} x\}} \quad (1)$$

forward slice of x includes all nodes influenced by x:

$$\boldsymbol{FS(x) = \{ y \mid x \rightarrow^{*} y \}} \quad (2)$$

Noninterference needs more information about security level of program statements. Thus, PDG is augmented by a security level lattice. In practice, it is sufficient to specify security level of input i.e. sources and output i.e. sinks statements. Sources and sinks are annotated by a so-called "provided security level" and "required security level" respectively. Provided security level P(x) means that x sends information with the provided security level or higher. Required security level R(x) specifies that information with smaller or equal security level can reach statement x.

Theorem 1 [11]: if
$\forall\, a \in dom(R) : \forall x \in BS(a) \cap dom(P) : P(x) \leq R(a)$
then confidentiality is maintained for all $x \in N$ (dom(P) and dom(R) represent set of sources and sinks respectively).

Based on theorem 1, information with a higher security level should never reach a sink with lower or non-comparable security level. Otherwise, a potential privacy leak is revealed ($P(x) \not\leq R(a)$). In this case, all responsible nodes are denoted by a so-called chop and are computed as follows:

$$\boldsymbol{Chop(x,a) = FS(x) \cap BS(a)} \quad (3)$$

C. IIFDroid Architecture

Fig. 4 presents a more detailed view of IIFDroid architecture. Initially, Android app is converted into Jimple bytecode via Dexpler [14]. Jimple is the main intermediate representation of Soot framework [15] which is appropriate for some specific analyses such as, points-to analysis, call graph and control flow graph construction (step 1-a).

To uncover privacy leaks between components, we need to consider inter-component communication. Accordingly, we leverage IC3 and IccTA in combination. IC3 takes Jimple representation as input and extracts all information about ICC methods like intents, URIs, intent-filters and target components. Besides, it stores all gathered information in a mySQL database (step 2-a).

In step 3-a, IccTA reads the database to build ICC links. Accordingly, IccTA instruments Jimple representation, connect components directly and obtains one component encompasses all components of the app. Additional analyses take place in this step. UI callbacks are collected through analyzing layout xml files. IIFDroid uses sources and sinks lists provided by SuSi tool [16] to find information sources and sinks in app. All application's potential entry points are extracted and used to construct a dummy main method as a unique entry point. This unique entry point is adopted to construct call graph of the app. At the end, ICFG of the app is constructed that shows how control transfers between different methods.

IIFDroid takes ICFG as input, build PDG of each procedure and connect them based on call and return

[3] Backward slices in SDG are calculated using a so-called HRB-slicing [13]- a context-sensitive slicing algorithm- that handles function calls properly.

relationship in ICFG to construct SDG of whole Android application (step 4-a). Then, it annotates SDG by provided security level of sources and required security level of sinks (step 4-b). It acts based on theorem 1, traverses SDG and calculates backward slice of sinks in a context-sensitive way to detect potential privacy leaks (step 4-c).

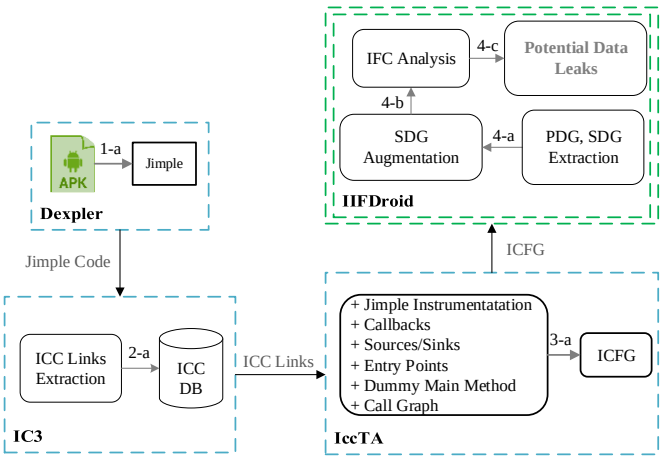

Fig. 4. An overview of IIFDroid Architecture

V. IMPLEMENTATION

IIFDroid extends Soot as its core framework. Soot is a Java static analysis framework which provides different intermediate representations, precise call graph and a raw PDG construction algorithm.

Jimple [15] is the base intermediate representation of Soot which IIFDroid uses. It is a typed, stackless, 3-address statement based intermediate representation that comprises only 15 instruction and is much simpler than Java and Android bytecode.

For precise modeling of app's lifecycle, IIFDroid acts like FlowDroid and IccTA by creating a dummy main method which emulates a unique entry point. The dummy main is taken as input by Spark, a precise call graph construction algorithm in Soot to build call graph of Android app. Afterwards, ICFG is constructed by IccTA and is given to IIFDroid as input.

IIFDroid starts with traversing ICFG and building PDG of each method encounters. The provided PDG in Soot is block-based and only contains control dependency between procedure's basic blocks. IIFDroid extends PDG construction process in 2 ways: (1) Constructs PDG with Jimple instructions granularity, (2) Adds flow-sensitive data dependence relationships between instructions.

Consider the code snippet in Fig. 5 that shows MainActivity class of DirectLeak1 test case in DroidBench. It reads IMEI in line. 10 and sends it via SMS.

We provided it as input to IIFDroid. The generated PDG is depicted in Fig. 6. IIFDroid annotated getDeviceId() with high as an information source and sendTextMessage() with low as a sink. It contains a path from sensitive source to a sink, transmitting information to outside the device. Thus, a security leak is reported.

```
1. public class MainActivity extends Activity {
2. @Override
3.
4. protected void onCreate(Bundle savedInstanceState){
5.   super.onCreate(savedInstanceState);
6.   setContentView(R.layout.activity_main);
7.   TelephonyManager mgr;
8.   mgr = (TelephonyManager)this.getSystemService(TELEPHONY_SERVICE);
9.   SmsManager sms = SmsManager.getDefault();
10.  sms.sendTextMessage("+49 1234", null, mgr.getDeviceId(), null, null);
11. }
12.}
```

Fig. 5: Direct Leak1

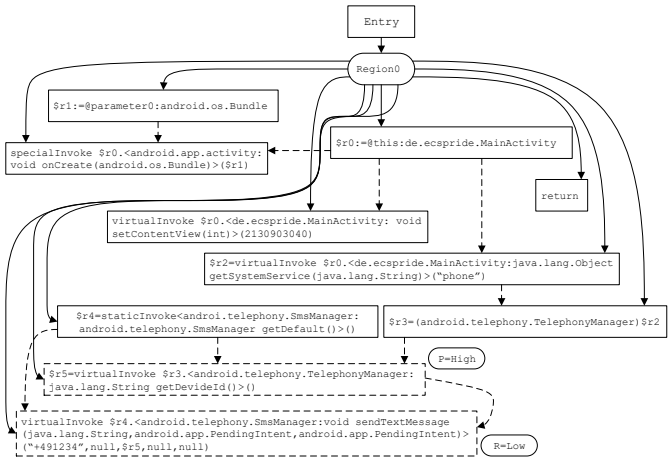

Fig. 6. PDG of DirectLeak1

VI. EVALUATION

We evaluated and compared IIFDroid with FlowDroid, IccTA and JoDroid on DroidBench and the developed test cases[4] to test for ICC and implicit flow leaks. The results are shown in Table I.

DroidBench [17] is an Android-specific test suite containing apps with known leaks which covers Android-specific aspects like callbacks, interactions with UI elements such as password fields, component's lifecycle, inter-component communications and Java-specific challenges like lists, arrays, reflection and static fields.

Additionally, there are some implicit flow test cases in DroidBench that only make use of if and switch structures. Thus, we have designed three test cases which leak IMEI using more complicated structures namely exception, polymorphism and throw like Fig. 2-b,d,e. They are specified with IIF-exception, IIF-Polymorphism and IIF-Throw in Table I.

Table II compares each tool characteristics and capabilities. For precise modeling of components lifecycle, four items have been defined. According to our findings, JoDroid only models Activity lifecycle properly. Moreover, it does not analyze layout xml files and is not capable of tracking UI information sources such as password fields.

FlowDroid overapproximates ICC communication and generates many false alarms. FlowDroid and IccTA are unable to discover implicit flows in sophisticated structures. Although JoDroid has been designed to guarantee noninterference in Android apps, it runs into many troubles

[4] github.com/zohreb71/Implicit-Flow-Test-Cases

in practice. It does not even find information sources and sinks properly within an Android app.

TABLE I. EVALUATION RESULTS
☑ = CORRECT WARNING, ☒ = FALSE WARNING, ⦸ = MISSED LEAK
MULTIPLE SYMBOLS IN ONE ROW: MULTIPLE LEAKS EXPECTED
ALL EMPTY ROW: NO LEAKS EXPECTED, NON REPORTED

Test Case	FlowDroid	IccTA	JoDroid	IIFDroid
DirectLeak1	☑	☑	⦸	☑
InativeActivity			☒	
LogNoLeak				
PrivateDataLeak1	☑	☑	⦸	☑
PrivateDataLeak2	☑	☑	⦸	☑
ArrayAccess1	☒	☒	☒	
ArrayAccess2	☒	☒		
ListAccess1	☒	☒		
AnonymousClass1	☑	☑	☑☒	☑
Button1	☑	☑	⦸	☑
Button2	☑☑☑☒	☑☑☑☒	☑☑⦸	☑☑☑
LocationLeak1	☑☑	☑☑	☑☑	☑☑
LocationLeak2	☑☑	☑☑	☑☑	☑☑
MethodOverride1	☑	☑	⦸	☑
FieldSensitivity1				
FieldSensitivity2				
FieldSensitivity3	☑	☑	⦸	☑
FieldSensitivity4				
ObjectSensitivity1				
ObjectSensitivity2				
Loop1	☑	☑	⦸	☑
Loop2	☑	☑	⦸	☑
SourceCodeSpecific1	☑	☑	⦸	☑
StartProcessWithSecret1	☑	☑	⦸	☑
StaticInitialization1	⦸	⦸	⦸	⦸
UnreachableCode			☒☒	
ActivityLifeCycle1	☑	☑	⦸	☑
ActivityLifeCycle2	☑	☑	⦸	☑
ActivityLifeCycle3	☑	☑	⦸	☑
ActivityLifeCycle4	☑	☑	⦸	☑
BroadcastReceiver-Lifecycle1	☑	☑	⦸	☑
ServiceLifecycle1	☑	☑	⦸	☑
Reflection1	☑	☑	⦸	☑
Reflection2	☑	☑	⦸	☑
Reflection3	☑	☑	⦸	☑
startActivity1	☑☒	☑	⦸	☑
startActivity2	☑☒☒☒☒	☑	⦸	☑
startActivity3	☑ (32☒)	☑	⦸	☑
startActivity4	☒☒			
startActivity5	☒☒			
startActivity6	☒☒			
startActivity7	☒☒	☒		☒
startActivityForResult1	☑	☑	⦸	☑
startActivityForResult2	☑	☑	⦸	☑
startActivityForResult3	☑☒	☑	⦸	☑
startActivityForResult4	☑☑☒	☑☑	⦸⦸	☑☑
startService1	☑☒	☑	⦸	☑
startService2	☑☒	☑	⦸	☑
bindService1	☑☒	☑	⦸	☑
bindService2	⦸	☑	⦸	☑
bindService3	⦸	☑	⦸	☑
bindService4	⦸☑☒	☑⦸	⦸⦸	☑⦸
sendBroadcast1	☑☒	☑	⦸	☑☒
insert1	⦸	☑	⦸	☑
delete1	⦸	☑	⦸	☑
update1	⦸	☑	⦸	☑
query1	⦸	☑	⦸	☑
Merge1	☒	☒		☒
ImplicitFlow1	☑	☑	⦸	☑
ImplicitFlow2	☑☑	☑☑	⦸⦸	☑☑
ImplicitFlow3	☑	☑	⦸	☑
ImplicitFlow4	☑☑	☑☑	⦸⦸	☑☑
IIF-Exception	⦸	⦸	⦸	☑
IIF-Throw	⦸	⦸	⦸	☑
IIF-Polymorphism	⦸	⦸	⦸	☑
Analysis Results				
Precision (☑/(☑+☒))	44.6%	89.6%	58.3%	94.8%
Recall (☑/(⦸+☑))	80.7%	91.2%	12.2%	96.4%
F-measure	0.57	0.9	0.2	0.95

All tools adopt flow and context-sensitive static analysis to uncover potential privacy leaks precisely. Finally, they are developed as an extension to existing Java static analysis frameworks, Soot and WALA.

TABLE II. TOOLS COMPARISON
✓ = FULL SUPPORT, ✷ = LIMITED SUPPORT, ×=NO SUPPORT

Criteria	FlowDroid	IccTA	JoDroid	IIFDroid
Activities	✓	✓	✓	✓
Services	✓	✓	×	✓
Content Providers	✓	✓	×	✓
Broadcast receivers	✓	✓	×	✓
UI Elements	✓	✓	×	✓
Inter-Component Communication	×	✓	×	✓
Implicit flows	✷	✷	×	✓
Precise detection of sources and sinks	✓	✓	×	✓
Flow and Context Sensitivity	✓	✓	✓	✓
Underlying Framework	Soot	Soot	WALA	Soot

CONCLUSIONS

In this paper, we have presented IIFDroid, a flow, context, object and field sensitive information flow control static analysis tool for Android apps which is able to uncover privacy leaks induced by explicit or different forms of implicit flows in an Andriod app. Our empirical findings show that it outperforms existing tools FlowDroid, IccTA and JoDroid with 94.8% precision and 96.4% recall.

Like existing tools, IIFDroid is unable to address static analysis challenges such as dynamic code loading, reflective call, native code and multithreading.

Privacy leaks are caused by different factors include extra app's permissions, advertisement libraries or developer mistakes. Nonetheless, IIFDroid only makes attempt to reveal potential privacy leaks in the most precise way. It does not look for the reason. All founded leaks are reported to the security analyst who can reason about them based upon auxiliary information such as app's permissions and functionalities.

Sometimes, Android apps co-operate to leak sensitive information. IIFDroid neglects inter-app communication and only models ICC within an application.

REFERENCES

[1] https://thenextweb.com/apps/2014/08/26/android-users-average-95-apps-installed-phones-according-yahoo-aviate-data/. [Accessed: May-2018].

[2] https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10. [Accessed: May-2018].

[3] L. Li et al., "Iccta: Detecting Inter-Component Privacy Leaks in Android apps", In *Proceedings of the 37th International Conference on Software Engineering*,Volume 1, pp. 280–291, 2015.

[4] B. Reaves et al., "* droid: Assessment and Evaluation of Android Application Analysis Tools", ACM Computing Surveys, vol. 49, no. 3, p. 55, 2016.

[5] https://developer.android.com/index.html. [Accessed: May-2018].

[6] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode", In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, p. 6, 2012.

[7] S. Arzt et al., "Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-aware Taint Analysis for Android Apps", Acm Sigplan Notices, vol. 49, no. 6, pp. 259–269, 2014.

[8] L. Li et al., "Static analysis of android apps: A systematic literature review", Information and Software Technology, vol. 88, pp. 67–95, 2017.

[9] W. You et al., "Android Implicit Information Flow Demystified", In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 585–590, 2015.

[10] D. Octeau et al., "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis", In *Proceedings of the 37th International Conference on Software Engineering*, vol.1, pp. 77–88, 2015.

[11] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs", International Journal of Information Security, vol. 8, no. 6, pp. 399–422, 2009.

[12] M. Mohr et al., "JoDroid: Adding Android Support to a static Information Flow Control Tool", In *Proceedings of CEUR Workshop*, vol.1337, pp. 140-145, 2015.

[13] S. Horwitz et al., "Interprocedural Slicing using Dependence Graphs", ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, pp. 26–60, 1990.

[14] A. Bartel et al, "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot", In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pp. 27–38, 2012.

[15] A. Einarsson and J. D. Nielsen, "A survivor's guide to Java program analysis with soot", BRICS Dep. Comput. Sci. Univ. Aarhus Den, p. 17, 2008.

[16] S. Rasthofer et al., "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks", In *Proceedings of 14th Network and Distributed System Securit (NDSS)*, 2014.

[17] "DroidBench benchmark", https://github.com/secure-software-engineering/DroidBench. [Accessed: May-2018].