Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

# An efficient approach for taint analysis of android applications

*Jie Zhang*[a,b], *Cong Tian*[a,b,*], *Zhenhua Duan*[a,b]

[a] *School of Computer Science and Technology, Xidian University, Xi'an 710071, P.R. China*
[b] *ICTT and ISN Laboratory, Xidian University, Xi'an, 710071, P.R. China*

ARTICLE INFO

*Article history:*
Received 17 July 2020
Revised 25 October 2020
Accepted 21 December 2020
Available online 24 December 2020

*Keywords:*
Android
Security
Static analysis
Taint analysis
Privacy

ABSTRACT

In recent years, sensitive data leaks of Android system attracted significant attention. The traditional facilities proposed for detecting these leaks, i.e. taint analysis, mostly focus on the precision and recall of the result with few of them addressing the importance of the cost and efficiency. As a matter of fact, the high costs of these tools often make them fail in analyzing large-scale apps and thus block them from wide usage in practice. In this paper, we propose FastDroid, an efficient and precise approach for taint analysis in Android apps with flow and context-sensitivity. First, upon groups of taint rules, a preliminary flow-insensitive taint analysis is conducted to construct the taint value graph which is an abstraction defined to describe the process of taint propagation in an app. Then, potential taint flows are extracted from the taint value graphs and further checked on the control flow graph to acquire the real taint flows. FastDroid is evaluated on the benchmark DroidBench, 1517 apps from Google Play store and 1022 apps from AndroZoo. The results show that the F-measure scores of FastDroid on DroidBench 2.0 and 3.0 are 0.89 and 0.75 respectively, the performance is better than the state-of-the-art tool FlowDroid. Further, a comparison on runtime with FlowDroid shows that FastDroid improves the efficiency significantly.



## 1. Introduction

Recent studies Matney (2018); Statcounte (2020) show that Android has over two billion monthly active users and dominates the smartphone market with a share of 74.1%, leaving other systems such as iOS, Windows mobile OS, and Blackberry far behind. The popularity also arouses the attention of the attackers who try to steal the sensitive data of users. Thus, Google continually introduces new security measures to Android systems such as runtime permission model (Android 6.0 and higher) Project (2019), Bouncer Kassner (2012) that is a security service for scanning the known malicious code in apps, and manually reviewing apps submitted to the Google Play Store PETROVAN (2015), etc. Nevertheless, the application security, especially the leak of sensitive data, is still a severe problem. Therefore, taint analysis for Android apps that addresses this issue attracts significant attention.

Sensitive data stored on the mobile phone can be utilized for diverse purposes. As a matter of fact, it is quite normal for apps accessing necessary sensitive data to realize their functions. For example, map apps request information of locations, and social apps access the uses' *Contacts*. However, if the usage of the sensitive data or the context where it leaks violates the users' privacy, it is regarded as a malicious behavior Enck et al. (2011). Hence, we believe that acquiring the whole picture of sensitive data leaks, which is the objective of taint analysis, is essential to help assess whether the leaks are malicious or legitimate.

* Corresponding author.
*E-mail addresses:* zhangjie@xidian.edu.cn (J. Zhang), ctian@mail.xidian.edu.cn (C. Tian), zhhduan@mail.xidian.edu.cn (Z. Duan).
https://doi.org/10.1016/j.cose.2020.102161
0167-4048/

Through the past few years, several taint analysis approaches have been proposed including both static and dynamic analyses Arzt et al. (2014); Enck et al. (2010); Feng et al. (2014); Gordon et al. (2015); Li et al. (2015); Lu et al. (2012); Wei et al. (2018). Our study focuses on the static analysis for two reasons. First, it has the advantage of covering all the code and performing a comprehensive analysis. Second, a malicious app could evade detection by concealing the illegal behavior in a test environment of dynamic analysis. Nevertheless, in static analysis, there is a trade-off between cost and precision, which generally depends on the extent of sensitivity that the approach supports, such as flow-sensitivity and context-sensitivity, as well as the algorithm and implementation that the approach utilizes Li et al. (2017). Normally, the more sensitivities a static analysis concerns, the more precise its result will be, but the cost correspondingly goes up. For example, the state-of-the-art tool FlowDroid Arzt et al. (2014) is one of the most remarkable approaches among them, which conducts a precise and context, flow, field, and object-sensitive taint analysis. FlowDroid makes a comprehensive modeling for the Android framework, implements the data flow tracking on the IFDS framework Reps et al. (1995). However, FlowDroid achieves a high precision and recall at the expense of the efficiency. In our tests, we find that FlowDroid is not quite efficient, especially for large-scale apps.

Google has announced that the size limitation of Android APK files is increased from 50MB to 100MB in 2015 to support richer apps and games Glick (2015). With the growth of the apps' size, the high costs of the analysis tools could be unacceptable or even fail in analyzing large-scale apps, and thus block them from wide usage in practice. As far as we know, the precise taint analyses are normally based on the traditional dataflow analysis. For example, FlowDroid utilizes the framework IFDS which is aimed to solve inter-procedural, finite, distributive, subset dataflow analysis problems Bodden (2012); Reps et al. (1995). Amandroid Wei et al. (2018) also implements the detection for the specific security problem on a data flow analysis framework. In our view, dataflow frameworks are heavy-weighted and designed for general purposes, the cost could be considerably high to solve specific problems, such as taint analysis. Hence, we aim to design a more efficient and precise approach tailored for taint analysis without using the dataflow framework.

In this paper, we present FastDroid, a novel static taint analysis tool that supports efficient taint analysis of apps in large scale, and meanwhile maintains a high precision and recall. Our strategy for taint analysis is quite different from the current precise analyses that based on the data flow analysis. We focus on exploring the propagation of taint values and construct taint value graphs (TVGs) to express the relationships of taint values in a flow-insensitive analysis. Then, we extract potential taint flows (PTFs) from the TVGs and check them on control flow graph (CFG) in order to prove the feasibility of PTFs and acquire the real taint flows. The feasibility checking is essential for our approach to attain flow- and context-sensitivity. FastDroid was first reported in the paper Zhang et al. (2019) and has been substantially extended since then.

To evaluate the performance of FastDroid, we have implemented a series of experiments on 1) DroidBench 2.0 and 3.0 GitHub (2016), benchmarks tailored for evaluating taint analysis on Android apps, 2) 1517 apps downloaded from Google Play store, and 3) 1022 apps from AndroZoo Allix et al. (2016). Compared with FlowDroid, FastDroid is better on precision and recall; meanwhile, it has a clearly smaller cost on time.

The contributions of this paper are summarized as follows:

- We propose a novel approach for taint analysis that significantly increases the efficiency and maintains a high precision and recall.
- We propose an abstraction TVG to express the relationships of taint values and define groups of taint rules to construct TVGs.
- We implement our approach and develop a tool FastDroid which is publicly accessible Fas (2020).

The rest of this paper is organized as follows. The next section briefly introduces taint flows and taint analysis. Section 3 shows a motivating example and the overview of our approach. Section 4 presents definitions of TVG and PTF, as well as several groups of taint rules. In Section 5, the proposed taint analysis approach is discussed in details. Section 6 presents the implementation and evaluation. Section 7 discusses the limitations of our work. Finally, Section 8 studies related work, and Section 9 concludes the paper.

## 2. Taint flows and taint analysis

### 2.1. *Taint flows in android app*

A taint flow describes the process of sensitive data leaking, which is normally composed of several taint values such as local or static variables, and fields of an object. It starts from a **source** that could be an API accessing the sensitive data (e.g., user's location) and ends at a **sink** that could be an API leaking the data (e.g., sending a short message) Rasthofer et al. (2014). Sensitive data is propagated through these taint values in various ways such as assignment, parameter passing, method returning, and aliases. If there exists a taint flow in an app and the relevant code is executed, the sensitive data will be leaked.

Fig. 1 shows an example of taint flow (abstracted from a real case). The life-cycle method *onCreate()* which is invoked whenever the activity is created contains a taint flow. The API *getDeviceID* that is a **source** returns the device identity number (sensitive data) which is stored in variable *a* (Line 3). Variable *c* is declared to be an alias of *a*, and thus becomes another taint value (Line 4). Variable *a* is passed to the method *function* as an argument. As a result, the parameter *x* is tainted. After a simple transformation of *x*, the string *y* is also tainted (Line 3 in b) and then returned to *b* (Line 5). Finally, *b* that stores the sensitive data is sent out the device via a short message by invoking the API *sendTextMessage* which is a **sink** (Line 6). To sum up, this taint flow starts from the invoking of *getDeviceID* to the invoking of *sendTextMessage* through the taint values of *a*, *x*, *y* and *b*.

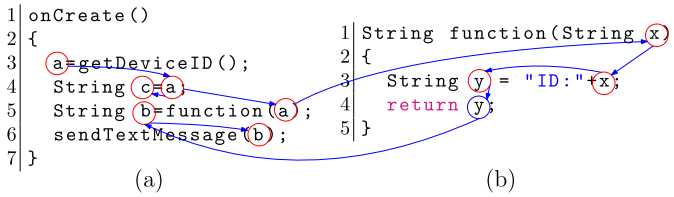


**Fig. 1 – An example taint flow. The circles mark all the taint values *a*, *c*, *x*, *y*, and *b*, and the curves express the process of taint propagation.**

### 2.2. *Challenges and current tools in static taint analysis*

There are several challenges in static taint analysis for Android apps: (1) Android apps are compiled to Dalvik bytecode rather than Java bytecode, which makes the original Java analyzing tools difficult to be utilized; (2) Android is an event-based system. There is no main method but multiple entry points in Android apps and the life-cycle methods are invoked by the Android framework; and (3) an app usually consists of several components that communicate with each other by using a special object called *Intent* Google (2019).

To cope with these challenges, various approaches for analyzing Android apps have been proposed these years. In the early stage, static analysis tools Barrera et al. (2010a); Enck et al. (2009); Felt et al. (2012) focus on the analysis of permission declared in the *Manifest* file. These tools are normally lightweight but imprecise, since the details of the code are ignored. An increasing number of approaches go beyond the permission level and push the analysis to code level. Some of the tools perform the analysis only on the abstractions of the code, such as call graph, APIs invoked or other predefined structures of the program Aafer et al. (2013); Felt et al. (2011); Gascon et al. (2013); Gibler et al. (2012). These methods improve the precision a lot, but only consider part of the program and few of them concern flow and context-sensitivity. So they are still not precise enough. In order to acquire a better precision and recall, the tools such as Arzt et al. (2014); Gordon et al. (2015); Li et al. (2015); Wei et al. (2018); Yang et al. (2015) build the analysis on the full pictures of the app (e.g., inter-procedural control flow graph) by modeling the Android environment comprehensively. These tools indeed improve precision a lot, but few of them put emphasis on efficiency. In practice, they can take quite a long time to track the sensitive data especially on large-scale apps.

## 3. Motivation and approach overview

### 3.1. *Motivation*

The current precise approaches usually trace the sensitive data along the paths on CFG. Meanwhile, it is essential to trace the aliases of taint values since the aliases are also taint values. Thus, we study the ways of alias analysis in different approaches. Amandroid Wei et al. (2018) which is aimed to provide a general static analysis framework for Android apps computes points-to information for all objects and their fields

```
1  void function()
2  {
3    a1=new A();//a1 is a heap object.
4    m1=new M();//m1 is a heap object.
5    a4=a3;
6    a2=a1;
7    m2=m1;
8    String s1=Source1();//Source 1
9    String s2=Source2();//Source 2
10   m1.s=s2;
11   m3=m2;
12   a1.s=s1;
13   a3=a2;
14   Sink1(a4.s);//Sink 1
15   Sink2(m3.s);//Sink 2
16 }
```

**Listing 1 – Example program with several aliases and two taint flows.**

before the data flow analysis. This strategy is costly and inefficient if the tool is only used for taint analysis, because most of the points-to information is useless for taint analysis. In contrast, FlowDroidArzt et al. (2014) combines a forward taint analysis with an on-demand backward analysis to search for the aliases of taint values Reps et al. (1995). The backward analysis is triggered whenever a taint value is assigned to a heap object. Throughout the backward analysis, once an alias of the taint value is found, a new forward analysis will be started from the point of alias. FlowDroid actually tracks each taint flow individually by a multi-threaded implementation Fritz et al. (2013). This strategy for alias analysis is more efficient than Amandroid since the on-demand feature saves much cost for acquiring points-to information of all objects Arzt (2017); Wei et al. (2018). But even so, when there exist numbers of taint heap objects or aliases, the cost of time and resources for backward and forward analyses could be considerably high.

For the example program shown in Listing 1, there exist two taint flows. According to the strategy of FlowDroid, to detect the "taint flow 1" that starts from *Source1*, a forward taint analysis is first triggered from line 8 since a string variable *s1* is tainted. In line 12, *s1* is assigned to a field of the heap object *a1* which has 3 aliases *a2*, *a3*, and *a4*. Thus, two backward analyses are triggered from lines 12 and 13, respectively. Also, another two forward analyses are started from lines 5 and 6, respectively. For the "taint flow2" that starts from *Source2*, there are totally two backward analyses and two forward analyses triggered. To sum up, there are five forward analyses and four

**Table 1 – The details of forward analysis (FA) and backward analysis (BA) in FlowDroid and the taint values detected in FastDroid.**

| | | | |
|---|---|---|---|
| **FlowDroid** | Taint Flow 1 | FA1 | 8,9,10,11,12,13,14,15 |
| | | BA1 | 12,11,10,9,8,7,6,5,4,3 |
| | | FA2 | 6,7,8,9,10,11,12,13,14,15 |
| | | BA2 | 13,12,11,10,9,8,7,6,5,4,3 |
| | | FA3 | 5,6,7,8,9,10,11,12,13,14 |
| | Taint Flow 2 | FA1 | 9,10,11,12,13,14,15 |
| | | BA1 | 10,9,8,7,6,5,4,3 |
| | | FA2 | 7,8,9,10,11,12,13,14,15 |
| | | BA2 | 11,10,9,8,7,6,5,4,3 |
| **FastDroid** | Iteration 1 | | *s*1, *s*2, *m*1.*s*, *a*1.*s* |
| | Iteration 2 | | *a*2.*s*, *m*2.*s*, *m*3.*s*, *a*3.*s* |
| | Iteration 3 | | *a*4.*s* |

backward analyses triggered when analyzing this program. Table 1 lists the lines of code to process in all these analyses.

### 3.2. *Approach overview*

FastDroid proposes a novel strategy for taint analysis, which combines the taint analysis and alias analysis together to improve efficiency. It focuses on the propagation of taint values, rather than carrying out the data flow analysis. Note that the aliases are treated as a taint propagative way in our approach. To be specific, it first performs a flow-insensitive taint analysis that traverses all the statements to detect the taint values and their aliases according to the predefined taint rules and constructs the TVGs which include the aliases and taint propagative information after several iterations. Then, potential taint flows are extracted from the TVGs and further checked on the CFG to prove their feasibility. For example, in Listing 1, FastDroid performs totally 3 iterations to detect all the taint values and processes 13 statements from line 3 to line 15 in each iteration. In the first iteration, two sources are found and the variables *s*1, *s*2, *m*1.*s*, and *a*1.*s* are detected as taint values. In the second iteration, the variables *a*2.*s*, *m*2.*s*, *m*3.*s*, and *a*3.*s* are found and *m*3.*s* are detected to be leaked, and "taint flow2" is determined. In the third iteration, the taint value *a*4.*s* is detected to be leaked, then "taint flow1" is determined. Totally, FastDroid processes 39 statements, much less than FlowDroid does. Table 1 lists the taint values and aliases explored in each iteration.

Based on the above taint analysis, FastDroid constructs the TVGs from which we can obtain two potential taint flows which are ($SC_1$, *s*1, *a*1.*s*, *a*2.*s*, *a*3.*s*, *a*4.*s*, $SK_1$) and ($SC_2$, *s*2, *m*1.*s*, *m*2.*s*, *m*3.*s*, $SK_2$). Here, $SC_1$ and $SC_2$ denote the sources, $SK_1$ and $SK_2$ denote the sinks. The other elements in the set are all taint values and the sequence of the values expresses the order of taint propagation. Finally, FastDroid searches realizable paths on the CFG through the relevant statements of each potential taint flow to prove the feasibility. For this example, both of the two potential taint flows are real taint flows.

FastDroid has an advantage of efficiency mainly due to the following three reasons: 1) there is no specific process for aliases searching such as FlowDroid's backward analyses which have the extra cost to initiate and control the backward and forward tracking, instead, the aliases are detected together with taint values in the iterations; 2) the preliminary taint analysis detects all the potential taint flows simultaneously; 3) although a further check for the potential taint flows is required, the search space is largely decreased because the statements referring to taint values have already been located.

## 4. Taint value graph and potential taint flow

In this section, we formally define the taint value graph and potential taint flow, which are utilized in our taint analysis, and explain the taint rules which are summarized from the taint propagative ways.

### 4.1. *Taint value graph*

We define an abstraction called taint value graph (TVG) which approximately expresses the taint propagative and alias relationships among the taint values to model the taint flows in an app. The taint value is a variable, or a field of an object or a class that holds the sensitive data at a certain period of time when the program is running. We use an access path to express a taint value as the form *x*.*a*.*b*. The *x* is a local variable or parameter or class, and *a* and *b* are fields Arzt et al. (2014). Access paths could have different lengths depending on the situation of taint values. As mentioned above, a taint flow starts from a **source**, ends at a **sink**, and propagates the sensitive data through a group of taint values. Thus, we define $TVG = (V, E)$ as a directed graph where a vertex in $V$ denotes a taint value, a source $SC_i$ or a sink $SK_j$, and an edge in $E$ represents a taint propagation, i.e. the source vertex taints the target vertex. Intuitively, a TVG consists of all the possible taint values derived from one **source** and the directed edges that express the taint propagation among these taint values. We may create multiple TVGs for an app depending on the number of the existing sources in the app.

According to the knowledge of taint propagation, we define three characteristics to construct a TVG: 1) there exists only one vertex namely $SC_i$ with the in-degree being 0 and there may exist several vertexes namely $SK_j$ with the out-degree being 0; 2) all the vertexes except $SC_i$ have at least one path from the vertex $SC_i$ to them; 3) there is no loop in the graph. The vertex with the 0 in-degree actually represents a source. As a source corresponds to a unique TVG, so there is only one $SC_i$ in a TVG. In addition, all the vertexes except $SC_i$ are taint values that tainted by the source or sinks that leak out the sensitive data, there must exist paths from $SC_i$ to them, since the taint values or sinks are all derived from the source. Besides, the reason why there exists no loop in the graph is that a value *A* could not be tainted by *A* itself or another taint value which is already tainted by *A* in reality.

We use graph structures to express taint flows rather than chain or tree structures due to the fact that the taint propagative ways among taint values are quite complex in reality. For example, if a method returns a taint value at different call sites and the returned value is assigned to different variables, the relationships of the returned taint value and the variables assigned to can be expressed in a branching mode. Moreover, if a method is invoked at different call sites with different taint

```
1 onCreate()
2 {
3   a=getDeviceID();//source
4   b=a;
5   c=function(a);
6   d=function(b);
7   sendTextMessage(c);//sink
8 }
```

(a)

```
1 String function(String x)
2 {
3   String y = "ID:"+x;
4   return y;
5 }
```

(b)

**Fig. 2 – Example program of branching and merging.**

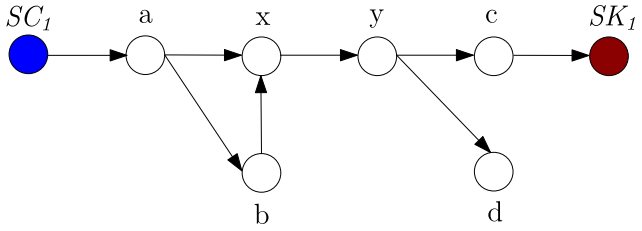


**Fig. 3 – The TVG of the example program.**

values passed into as arguments, a merging mode is necessary to express the relationships of the taint values and the formal parameter. Take the program in Fig. 2 as an example. The method *function* is invoked in the life-cycle method *onCreate*, and two different taint values *a* and *b* are passed into the formal parameter x. In this case, we utilize a merging mode in the TVG to express that x is tainted by both a and b. Then, the method *function* returns a taint value y that is assigned to the variables *c* and *d* which are both tainted. Here, a branching mode can express the relationships among these three taint values. The TVG of this program is depicted in Fig. 3.

### 4.2. *Potential taint flow*

From TVGs, we extract the paths from $SC_i$ to $SK_j$ for generating the potential taint flows (PTFs). A PTF formalizes a possible taint flow from the view of taint propagation. We use a variant of the path in TVG to express a PTF, which is an ordered set composed of ordered pairs. An ordered pair actually expresses a taint propagation of two taint values, and the sequence of ordered pairs expresses the sequence of the propagation. Since an app may contain multiple taint flows, there could exist a number of PTFs in an app. Formally, a PTF $PTF_i$ is defined below:

$$PTF_i = \{< SC_i, X_1 >, < X_1, X_2 >, \cdots, < X_n, SK_j >\}$$

where $SC_i$ is a source, $SK_j$ is a sink, $X_i (1 \leq i \leq n)$ is an intermediate taint value. The ordered pair $< X_{i-1}, X_i >$ denotes that the latter taint value is tainted by the former one (e.g. an assignment statement $X_i = X_{i-1}$). The ordered pairs $< SC_i, X_1 >$ and $< X_n, SK_j >$ denote that $X_1$ is tainted by a source $SC_i$ and $X_n$ is leaked out by a sink $SK_j$, respectively. Since FastDroid generates the TVG from an imprecise flow-insensitive taint analysis, the PTFs created may include both real and spurious taint flows. This is the reason why the PTFs are called "potential" taint flows. For example, there are two PTFs extracted from the TVG in Fig. 3, which are $\{< SC_1, a >, < a, x >, < x, y >, < y, c >, < c, SK_1 >\}$ and $\{< SC_1, a >, < a, b >, < b, x >< x, y >, < y, c >, < c, SK_1 >\}$. But the latter one does not exist in practice and should be eliminated in the further checking step.

**Table 2 – Taint Rules.**

*Source Rule*
R1. $< SC, x >$:- $S(x = m()), m \in Source$.
*Propagation Rules*
R2. $< x.*, y.* >$:- $S(y = x), IN(x.*)$.
R3. $< x.*, y.* >$:- $S(x = y), hp(y), IN(x.*)$.
R4. $< x.*, y.* >$:- $S(y = m()), Ret(m, x.*)$.
R5. $< x_i.*, p_i.* >$:- $S(p_i = m.para_i), Para(m, i, x_i.*)$.
R6. $< p_i.+, x_i.+ >$:- $S(m(x_0, \ldots, x_n)), hp(x_i), Ret(m, i, p_i.+)$.
R7. $< p_i.+, x_i.+ >$:- $S(y = m(x_0, \ldots, x_n)), hp(x_i), Ret(m, i, p_i.+)$.
R8. $< x, y >$:- $S(y = m(x)), m \in TW, IN(x)$.
*Method Rules*
R9. $Ret(m, x.*)$:- $S(return\ x), IN(x.*)$.
R10. $Ret(m, i, p_i.+)$:- $S(p_i = m.para_i), hp(p_i), IN(p_i.+)$.
R11. $Para(m, i, x_i.*)$:- $S(m(x_0, \ldots, x_n)), IN(x_i.*)$.
R12. $Para(m, i, x_i.*)$:- $S(y = m(x_0, \ldots, x_n)), IN(x_i.*)$.
*Sink Rules*
R13. $< y, SK >$:- $S(x = m(., y, .)), IN(y), m \in Sink$.
R14. $< y, SK >$:- $S(m(., y, .)), IN(y), m \in Sink$.
*Cleaning Rules*
R15. $CTV(x)$:- $S(x = y), IN(x), \neg IN(y)$.
R16. $CTV(x)$:- $S(x = Null), IN(x)$.
*Implicit Rule*
R17. $ITV(x)$:- $S(condition(x)), IN(x)$.

### 4.3. *Taint rules*

To construct the TVG, we summarize taint propagative ways that we know and define groups of taint rules. In the rules, the alias of taint value is also regarded as a kind of taint propagation. FastDroid utilizes these taint rules to determine whether a statement generates a new vertex of the TVG or not. We express the taint rules in a language similar to Datalog language Huang et al. (2011), which can describe a high level but complex algorithms in an easy way. Table 2 shows the taint rules which are divided into 6 groups. Each rule consists of two parts separated by the: - symbol. If all the predicates on the right side are satisfied, the relation or predicate on the left side is true. A relation expresses that the former value taints the latter one. For instance, the relation $< x, y >$ means that the variable x taints the variable y. The predicate S() specifies the type of the statement S which could be assignment, declaration, invoking, returning, or conditional statement. The predicate $IN(x)$ means that the variable x is already involved in the taint values (the vertex set of a TVG). The predicate $hp(y)$ denotes that the variable y is a heap object.

```
1 void function()
2 {
3   String number=getDeviceID();//Source
4   if(number.equals("133********"))
5       sendTextMessage("got it!");
6 }
```

**Listing 2 – Example program of implicit taint propagation.**

Specifically, the rule *R1* in the source rule group expresses the way to find a source. If *S* is an assignment statement where the left value is *x* and the right value is a call to an API method belonging to the *Source* (a set of all the source API methods), the relation $< SC, x >$ is true (*SC* denotes the source). The propagation rules group includes the rules about the taint propagation. *R2* says that if an assignment statement *S* has a left value *y* and a right value *x* that is tainted (a taint value or has a tainted field), then $< x.*, y.* >$ is true ($x.*$ denotes a variable x or a field of x). *R3* is about the alias: if a heap object *y* has an alias x which is tainted or contains a tainted field, $< x.*, y.* >$ is true.

To explain the rules $R4 - R7$, we first introduce several predicates in the method rules that express the inter-procedural taint propagation. The rule *R9* says that if a return statement *S* returns a tainted value *x*, the predicate $Ret(m, x.*)$ is true. This means that the method *m* returns a taint value $x.*$ at certain call sites. *R10* defines a predicate $Ret(m, i, p_i.+)$ that is true when $p_i$ (the $i_{th}$ formal parameter of *m*) is a heap object and has a field tainted in *m* ($p_i.+$ denotes a field of $p_i$). The rules $R11 - R12$ define how the parameters are tainted. If *S* invokes a method *m* which is passed into a tainted variable *x* as the $i_{th}$ argument, then the left predicate $Para(m, i, x_i.*)$ is true. Now, we return to explain $R4 - R7$. Rule *R4* says that if *S* is an assignment statement and the left value is *y*, and the right value is a call to *m* that satisfies $Ret(m, x.*)$, then $< x.*, y.* >$ is true. The rule *R5* expresses that if $Para(m, i, x_i.*)$ is true, the $i_{th}$ formal parameter $p_i$ or its field is tainted by the argument $x_i.*$. $R6 - R7$ define that if the method m following $Ret(m, i, p_i.+)$ is invoked, the corresponding field of $i_{th}$ argument *x* is tainted by the taint value $p_i.+$.

Since it is tedious and costly to analyze taint flows through the API methods of JRE and Android SDK, we define shortcuts to accelerate this process. We build a library of taint wrapper methods *TW* to collect the API methods in JRE and Android SDK that directly return taint values if they are passed into taint values as arguments (e.g. *toString*()). This mechanism is expressed in the rule *R8*: when a method *m* in *TW* obtains a tainted argument *x*, it returns a taint value and taints the left value *y*. The rules *R13* and *R14* define the way for detecting the sinks. If a method belongs to *Sink* (a set of all the sink API methods) and is invoked with a tainted argument *y*, the relation $< y, SK >$ is true (*SK* represents the sink).

The rules *R15* and *R16* are special rules for checking PTFs. They describe a taint value *x* can be *cleaned* by being assigned an untainted value *y* or a null value. The cleaned taint values are recorded into a list *CTV*. The rule *R17* is summarized according to the cases of implicit taint propagation. As shown in Listing 2, the variable *number* holds sensitive data (Line 3) and is used in the conditional statement (Line 4). Although the sensitive data is not directly leaked out, another signal is sent out to imply the data. FastDroid handles this situation with the implicit rule *R17*: whenever a taint value is used in a conditional statement, it is regarded as being leaked implicitly. We record these kind of taint values in a list *ITV* to provide a reference of sensitive data leaks.

## 5. Taint analysis

In this section, we explain in detail how the taint analysis of FastDroid works, how the TVGs are constructed and PTFs are extracted from them, why and how the feasibility of PTFs are checked on the CFG.

### 5.1. *Constructing taint value graphs*

FastDroid performs a flow-insensitive taint analysis to construct TVGs based on a tree structure of the app rather than on the CFG. We use the Jimple code Vallee-Rai and Hendren. (1998) to create the tree structure. The root of the tree consists of a set of classes (SC) that involves all the classes in the app. Each class (C) has a set of methods and each method (M) contains a set of statements (S) that forms the leaves of the tree. The leaves of the tree are organized according to the default sequence of the Jimple statements.

FastDroid implements Algorithm 1 to construct the taint value graphs, which processes every leaf of the tree structure

**Algorithm 1:** Constructing taint value graphs.

**Input**: Tree structure of an app
**Output**: A set of all *TVG*s, *CTV*s and *ITV*s: *TVGSet*

```
1  TVGSet = ∅;
2  p = 0; q = 0;
3  while TVGSet is initialized or updated do
4      for each C ∈ SC, M ∈ C, and S ∈ M do
5          if S satisfies R1 then
6              V_p = {SC_p, x}; E_p = {< SC_p, x >};
7              create a new TVG_p = (V_p, E_p);
8              create a new CTV_p = ∅;
9              create a new ITV_p = ∅;
10             add (TVG_p, CTV_p, ITV_p) into TVGSet;
11             p = p + 1;
12         if S satisfies R2 − R8, x ∈ V_i in TVG_i then
13             add y into V_i; add < x, y > into E_i;
14         if S satisfies R9 − R12 then
15             create corresponding Ret() or Para();
16         if S satisfies R13 − R14, y ∈ V_i in TVG_i then
17             add SK_q into V_i; add < y, SK_q > into E_i;
18             q = q + 1;
19         if S satisfies R15 − R16, x ∈ V_i in TVG_i then
20             add x into CTV_i;
21         if S satisfies R17, x ∈ V_i in TVG_i then
22             add x into ITV_i;
23 return TVGSet
```

with the taint rules. The input of Algorithm 1 is the tree structure of an app; the output is a set *TVGSet* that contains all the *TVG*s as well as their related *CTV*s and *ITV*s. In lines 1–2, *TVGSet* is initialized as an empty set and the variables $p$ and $q$ are initialized as 0, which are respectively used to count the sources and sinks detected. In line 3, the *while* loop starts the construction of *TVG*s and it will continue if the *TVGSet* is just initialized or updated in the last iteration to make sure all the *TVG*s are completed. In line 4, the *for* loop traverses the statements which are the leaf on the tree to detected the sources, sinks and taint values with taint rules. If any rule is satisfied, the corresponding operations are performed according to the rule.

Specifically, if a statement $S$ satisfies the source rule $R1$, a new $TVG_p$ is created, which includes the vertexes $SC_p$, $x$ and the edge $< SC_p, x >$. Then the $TVG_p$ and the related $CTV_p$ and $ITV_p$ initialized as empty sets are added into the *TVGSet* as elements (Lines 5–11). While if $S$ follows one of $R2 - R8$, the variable $x$ is assumed to be an existing taint value in $TVG_i$ and the relation $< x, y >$ is satisfied, then the new vertex $y$ is added into $V_i$ and the new edge $< x, y >$ is added into $E_i$ (Lines 12–13). If $S$ satisfies one of $R9 - R12$, the corresponding predicate is created (Lines 14–15). If $S$ satisfies $R13$ or $R14$ and the taint value $y$ in $TVG_i$ is leaked out, which means a sink is found, the vertexes $y$, $SK_q$ and the edge $< y, SK_q >$ are added into $TVG_i$ (Lines 16–18). Furthermore, if $S$ satisfies $R15$, $R16$ or $R17$, the related taint value in $TVG_i$ is added into the list $CTV_i$ or $ITV_i$ (Lines 19–22).

The *while* loop in Algorithm 1 restarts another iteration when any new element is added into *TVG*s in the last iteration. The set *TVGSet* is expanded gradually in each iteration until no more update occurs. The running time of the algorithm apparently depends the number of statements $N$ and the number of iterations. The number of iterations depends on the number of vertexes (taint values) in TVGs as well as their taint ways. In general, the more aliases and functions that the taint flows related to, the more iterations are required. We assume that the TVG $TVG_{max}$ has the most vertexes in all TVGs and $V$ is the number of the vertexes. In the extreme case, the taint values in $TVG_{max}$ are all aliases or tainted through different functions. The number of iterations is at most $V$ and the running time of Algorithm 1 is $O(VN)$. However, in most cases, taint flows are not that complicated. In the simplest case, if the taint flow is just located in a single method and contains no alias, and meanwhile the detection for the taint values follows the default sequence of the statements. Here the runtime reduces to $O(N)$.

### 5.2. *Checking potential taint flows*

FastDroid extracts PTFs from the TVGs by exploring the paths from the sources to the sinks on the TVGs. FastDroid formalizes the paths as PTFs and checks their feasibility on the CFG. After the checking, the real taint flows in PTFs are proved and the spurious taint flows in PTFs are eliminated. In the following, we first clarify two issues about FastDroid and then explain the checking process:

***Why is it necessary to check the feasibility of PTFs?***

The previous imprecise flow-insensitive analysis creates the TVGs and generates any possible taint values according to the taint rules without the information of control flows or call-sites. In this process, the TVGs may include some uncertain taint values which make the PTFs imprecise and result in spurious taint flows. Therefore, the feasibility checking process is crucial to ensure the soundness of our approach. We enumerate some common cases where spurious taint flows occur: 1) there does not exist any path on the CFG passing through the relevant statements of PTF, e.g., the sink executes before the source; 2) the taint value is cleaned on the path before tainting the next value, e.g., the taint value is assigned a null value; 3) some statements related to the PTF can never be reached from the starting point of the program and can never be executed.

***How does FastDroid maintain flow and context-sensitivity?*** A flow-sensitive taint analysis takes into account the execution order of statements. In the checking process of PTFs, all the related statements of PTFs are located on real execution paths of CFG, and their sequences are considered. Besides, the context-sensitivity requires the analysis to record the information of call-sites when functions are invoked or returned. In the construction of TVGs, we have saved the location and context for each taint value detected, which include the information of call-sites. Moreover, the checking of PTFs performs an inter-procedural analysis which concerns the call-site issue.

The checking process consists of the following steps: 1) CFG construction, 2) generating expected paths of PTFs, and 3) searching the realizable execution paths.

***CFG construction.*** There are several current facilities that already provide solutions for constructing precise CFG of Android apps as a foundation of data flow analysis Ammons and Larus (1998). FastDroid utilizes the CFG constructed by the tool FlowDroid 2.0 which creates a dummy main method to invoke the life-cycle and callback methods to model the Android runtime. In addition, FlowDroid 2.0 integrates in an ICC analysis module IccTA that creates several ICC helper methods to replace the ICC methods (e.g. startActivity). As a result, FastDroid also supports ICC analysis by applying the method rules to the ICC helper methods.

***Generating expected paths of PTFs.*** An ordered pair in a PTF is derived from a taint propagation caused by a statement. Thus, we can deduce a sequence of statements from a PTF with the same sequence of the order pairs, called the *expected path* of a PTF. An expected path is not a real execution path on CFG but to reflect the taint propagation. We consider the relationship between these two paths. In general, if an expected path $ep_i$ completely **matches** an execution path (the statements on two paths are in the same order), then the relevant PTF $PTF_i$ is feasible and is proved to be a real taint flow.

However, there are two exceptions due to the alias and the "cleaning" operation. As we define in Section 4, if a taint value is generated from the rule $R3$, which means it is an alias of an existing taint value $TV_e$, then the declaration of the alias must be ahead of the statement that taints the $TV_e$, otherwise this taint propagation of the alias is invalid. To express this conflict of expected path concerning to aliases, a flag "$\leftarrow$" is marked to the declarative statement of the alias so as to indicate this statement must be placed backward. Besides, for a cleaned taint value $TV_c$ generated from the rule $R15$ or $R16$, the statement performing cleaning operation $Stmt_c$ should not occur during the statement $Stmt_t$ where $TV_c$ is tainted and

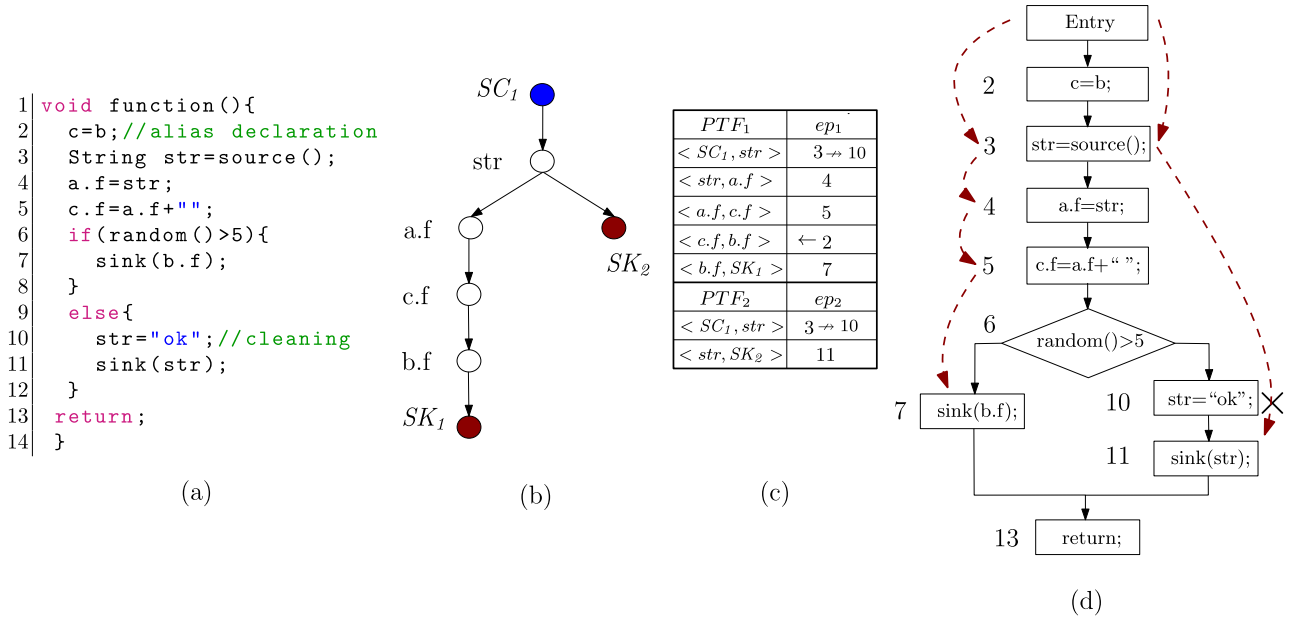


**Fig. 4 – Example of the taint analysis process. For the example program in (a), we can construct the TVG depicted in (b), from which two PTFs can be extracted, namely $PTF_1$ and $PTF_2$. The PTFs and their related expected paths are listed in (c). (d) shows the checking process of the two expected paths.**

the statement where $TV_c$ taints other variables, because it has been cleaned in this period and the taint flow is already cut off. In this case, we mark the statement $Stmt_t$ with a flag "$\nrightarrow$" as well as the location of $Stmt_c$, to indicate $Stmt_c$ could not follow $Stmt_t$. In our taint analysis, the information of taint values related to alias and cleaning are recorded, we only make a simple transformation for PTFs to obtain the *marked* expected paths.

The example program in Fig. 4 (a) includes both the above two exceptions. Fig. 4 shows the TVG of the program, two PTFs extracted from the TVGs and the ordered pairs in the PTFs. The expected paths of the PTFs are $ep_1 = (3, 4, 5, 2, 7)$ and $ep_2 = (3, 11)$. For $PTF_1$, the taint value *b* tainted at the statement 2 is an alias of the taint value *c*. So we mark the statement 2 in $ep_1$ with "$\leftarrow$" to indicate that the statement 2 must be executed before the previous statement 5. Besides, the taint value *str* is cleaned at the statement 10. So we mark the statement 3 that taints *str* with "$\nrightarrow 10$" to indicate that the statement 10 could not occur just following the statement 3. Then $ep_1$ turns to be $(3 \nrightarrow 10, 4, 5, \leftarrow 2, 7)$, and $ep_2$ turns to be $(3 \nrightarrow 10, 11)$.

***Searching the realizable execution paths.*** FastDroid runs Algorithm 2 to search all the realizable execution paths that start from the entry of the app and pass through the expected paths in sequence with concern of the two exceptions (cleaning and aliases). Depending on the fact whether the realizable paths exist, the real taint flows are proved and the spurious taint flows are eliminated. Since the relevant statements of PTFs are already located, the search space of statements is actually reduced a lot compared with the entire code.

Algorithm 2 takes the expected path of a PTF and CFG of the app as inputs. The output is a boolean value to express the feasibility of the PTF. The algorithm starts with a set of initializations (Lines 1-2). $S_c$ and $S_d$ denote the start and end statements in each iteration. $P_{sum}$ is used to store the currently

**Algorithm 2:** Checking expected path of PTF.

**Input**: $ep = (S_1, S_2, \ldots, S_n)$, *CFG*
**Output**: the PTF is feasible or not (**true** or **false**)

```
1  Sc = Entry_main; i = 1; Sd = Si;
2  cleanS = null; Psum = ∅;
3  while i ≤ n do
4      if Sd is marked with "←" then
5          if Sd ∉ Psum then
6              return false
7          else
8              eliminate the paths without Sd from Psum ;
9              i++; Sd = Si;
10     else
11         if ∃ paths Path from Sc to Sd on CFG then
12             for each path in Path do
13                 if cleanS == null or cleanS ∉ path then
14                     add path into Psum;
15             if Psum is added then
16                 Sc = Sd; i++; Sd = Si;
17             else
18                 return false;
19             if Sd is marked with "↛ X" then
20                 cleanS = X;
21             else
22                 cleanS = null;
23         else
24             return false;
25 return true;
```

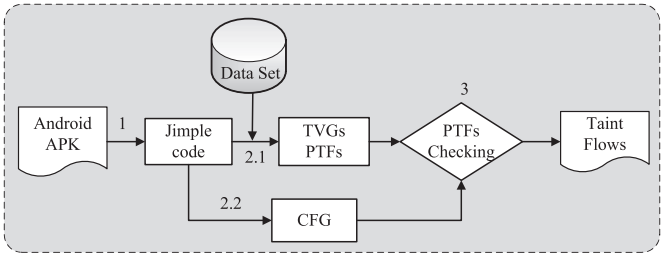


**Fig. 5 – Overview of FastDroid: the input is an APK file and the output is the taint flows.**

obtained realizable paths. *cleanS* records the statement cleaning the current taint value. Then, a while loop is carried out to search paths on the *CFG* through the expected path *ep* (Lines 3-24). In each iteration, the algorithm checks a segment of *ep* from $S_c$ to $S_d$. If $S_d$ is marked with "←", then the algorithm checks if $S_d$ occurs on any paths in $P_{sum}$. If so, the paths without $S_d$ are eliminated and $S_d$ moves to the next statement in *ep* (Lines 7-9). If no path contains $S_d$, the algorithm returns **false** directly which means the expected path is unrealizable (Lines 5-6). If the $S_d$ is not marked with "←", the algorithm searches all execution paths from $S_c$ to $S_d$. Each such path is inspected to make sure that *cleanS* is not involved in, and only the paths without *cleanS* are added into $P_{sum}$ (Lines 12-14). If $P_{sum}$ is added at least one path, $S_c$ moves to $S_d$ and $S_d$ moves to the next statement (Lines 15-16). Otherwise, the algorithm returns **false**. Here, for the statement marked with "$\nrightarrow X$" which means the statement $X$ cleans the current taint value, $X$ is recorded in variable *cleanS* for further inspection (Lines 19-22). In the end, the algorithm returns **true** if the checking procedure reaches to $S_n$, which indicates the PTF is feasible.

For instance, in Fig. 4 (d), $ep_1$ is checked to be feasible on the execution path (2, 3, 4, 5, 6, 7), where the statement 2 marked with "←" occurs prior to statement 5 and statement 10 does not occur on the path. However, there is no path that matches $ep_2$, since the only path (2, 3, 4, 5, 6, 10, 11) passing through the statement 11 executes the statement 10 between statement 3 and 11, and the taint flow is cut off. Consequently, $PTF_1$ is a real taint flow but $PTF_2$ is a spurious taint flow.

## 6. Implementation and evaluation

In this section, we illustrate the implementation details and evaluation of FastDroid. Our implementation consists of about 8000 lines of Java code and leverages several modules of FlowDroid 2.0 GitHub (2018) which is built upon a number of publicly available tools such as Soot Lam et al. (2011), IC3 Octeau et al. (2015), Dexpler Bartel et al. (2012), IccTA Li et al. (2015). The tool FastDroid is publicly accessible from the website Fas (2020). Fig. 5 shows an overview of FastDroid. In Step 1, Soot converts the Android APK file into *Jimple* code. In Step 2.1, FastDroid performs a preliminary flow-insensitive taint analysis on the *Jimple* code to construct the TVGs and the PTFs with the help of a date set that stores the source and sink methods; and in Step 2.2, some modules of FlowDroid are utilized to generate the CFG which includes the dummy main methods and ICC helper methods. In Step 3, FastDroid checks all the PTFs on the CFG and finally we obtain the final taint flows.

To evaluate the performance of FastDroid, we conducte a serials of experiments and compare the precision, recall, and efficiency with FlowDroid 1.0 and 2.0 which are set as the default configurations. We use three test suites: 1) DroidBench 2.0 and 3.0 Arzt et al. (2014); Pauck et al. (2018), well-known benchmarks tailored for the taint analysis of Android apps, 2) 1517 apps randomly downloaded from Google-Play store, and 3) 1022 apps[1] from AndroZoo dataset Allix et al. (2016) that are available in popular app stores such as Google Play and Anzhi Market. In the empirical study of Luo et al. Luo et al. (2019), these 1022 apps from AndroZoo have been analyzed with FlowDroid and reported to contain taint flows. We conducted all the experiments on a machine with 4.0 GHz Intel Core i7 CPU and 16 GB memory.

### 6.1. *Effectiveness*

DroidBench 2.0 and 3.0 respectively contain 119 and 190 open-source apps[2] with specified sets of taint flows to be found. In Table 5, we list the detailed result of the experiment on DroidBench 2.0 to evaluate the effectiveness of these three tools. The result of FlowDroid 1.0 achieves a precision of 84.9% and a recall of 64.6%, while FlowDroid 2.0 achieves a precision of 84.7% and a recall of 73.5%. Note that FlowDroid 1.0 misses a lot of leaks in the Inter-Component-Communication category, but FlowDroid 2.0 detects most of them as the IccTA module is integrated into the tool. Compared with them, FastDroid achieves both the best precision of 93.3% and the best recall of 85.8%. Furthermore, the F-measure of FastDroid which is a combined measure for precision and recall, has the highest value of 0.89. For the category of Implicit-Flows, FastDroid has the implicit rule for handling them and gives correct warnings. Since FlowDroid 2.0 obviously performs better than version 1.0, in the following statistics we only make comparison between FlowDroid 2.0 and FastDroid.

In Table 3, we count the correct warnings, false warnings and missed leaks reported by FlowDroid 2.0 and FastDroid in all categories of DroidBench 3.0. Both of the tools are not able to detect the taint flows in categories including Dynamic-loading, Native, IAC (Inter-App-Communication),

[1] https://www.kaggle.com/covaanalyst1/cova-dataset.
[2] https://github.com/FoelliX/ReproDroid.

**Table 3 – Test Results on DroidBench 3.0. FLD: FlowDroid 2.0, FastD: FastDroid, C: correct warning, M: missed leak, F: false warning.**

| Tools | | FLD | | | FastD | | |
|---|---|---|---|---|---|---|---|
| ID | Category | C | M | F | C | M | F |
| 1 | Aliasing | 1 | 2 | 1 | 2 | 1 | 1 |
| 2 | AndroidSpecific | 7 | 4 | 1 | 7 | 4 | 0 |
| 3 | ArraysAndLists | 4 | 0 | 4 | 4 | 0 | 1 |
| 4 | Callbacks | 15 | 2 | 3 | 17 | 0 | 1 |
| 5 | DynamicLoading | 0 | 3 | 0 | 0 | 3 | 0 |
| 6 | EmulatorDetection | 14 | 4 | 0 | 17 | 1 | 0 |
| 7 | Field&ObjectSens | 2 | 0 | 0 | 2 | 0 | 0 |
| 8 | GeneralJava | 17 | 7 | 4 | 20 | 4 | 2 |
| 9 | ImplicitFlows | 0 | 6 | 0 | 4 | 2 | 0 |
| 10 | IAC | 0 | 15 | 0 | 0 | 15 | 0 |
| 11 | ICC | 14 | 5 | 1 | 15 | 4 | 0 |
| 12 | Lifecycle | 20 | 4 | 0 | 22 | 2 | 2 |
| 13 | Native | 0 | 5 | 0 | 0 | 5 | 0 |
| 14 | Reflection | 1 | 8 | 0 | 1 | 8 | 0 |
| 15 | Reflection_ICC | 0 | 11 | 0 | 0 | 11 | 0 |
| 16 | SelfModification | 0 | 3 | 0 | 0 | 3 | 0 |
| 17 | Threading | 6 | 0 | 0 | 3 | 3 | 0 |
| 18 | UnreachableCode | 0 | 0 | 3 | 0 | 0 | 3 |
| **Sum, Precision and Recall** | | | | | | | |
| **Total** | | 101 | 79 | 17 | 114 | 66 | 10 |
| **Precision** $p = C/(C+F)$ | | 85.6% | | | 91.9% | | |
| **Recall** $r = C/(C+M)$ | | 56.1% | | | 63.3% | | |
| **F-measure** $2pr/(p+r)$ | | 0.68 | | | 0.75 | | |

**Table 4 – Test Results on Google-Play and AndroZoo. FLD: FlowDroid 2.0, FastD: FastDroid.**

| | Google-Play | | AndroZoo | |
|---|---|---|---|---|
| **Tools** | **FLD** | **FastD** | **FLD** | **FastD** |
| **Apps Counts** | 1517 | 1517 | 1022 | 1022 |
| **Apps with Leaks** | 1104 | 1238 | 981 | 990 |
| **Leaks Counts** | 26,644 | 17,908 | 15238 | 20618 |
| **Average Time (s)** | 67.99 | 6.20 | 57.1 | 13.7 |
| **Max Time (s)** | 458.83 | 292.34 | 522.1 | 413.6 |
| **Min Time (s)** | 0.09 | 0.01 | 0.63 | 0.07 |

Reflection_ICC and Self-modification. However, in the groups of IAC and Reflection_ICC, both of tools actually report partial taint flows which only include sections of existing taint flows in an individual app or component. For example, a partial taint flow starts from a source method *getStringExtra()* of an *Intent* object or leaks at a sink such as *startActivity()*. In our view, although these taint flows are not precise, they reflect parts of the data leaks, and thus we do not regard them as false warnings. As the result shows, the precision and recall of FastDroid are both better than FlowDroid on DroidBench 3.0. The F-measure of FastDroid achieves 0.75.

Table 4 presents the results on Google-Play and AndroZoo. For 1517 apps in Google-Play, FastDroid detects 1238 apps with 17,908 taint flows, while FlowDroid detects 1104 apps with totally 26,644 taint flows. Besides, we observe that 1042 apps with at least one taint flow are detected out in both tools. For AndroZoo, although the previous study shows that FlowDroid reports data leaks for all these apps, we only detect 981 apps with totally 15,238 taint flows with FlowDroid in our experiment. This is probably because the configuration or the files of sources and sinks are different. Whereas, FastDroid detects 990 apps with 20,618 taint flows. Among them, 958 apps are reported in both tools that contain taint flows. Note that we do not evaluate the precision or recall of the experiments on Google-Play and AndroZoo, since there is no ground truth of the taint flows on these real-world apps and it is difficult to make a manual reverse engineering.

### 6.2. *Efficiency*

We evaluate the runtime performance of FastDroid and FlowDroid 2.0 on DroidBench, Google-Play and AndroZoo apps. We believe that the memory size of the APK file alone is not a good measure for the scale of the app, because the file could also involve in images, audio and video files. Thus, we evaluate the tools from the two aspects of code size and memory size. For each app, we count the number of Jimple statements in the app as an approximate measure of the code size of the app. Besides, since FastDroid and FlowDroid contain the same procedure for CFG construction, we only count in the runtime spent on taint analysis without the preparation procedure for CFG construction.

For DroidBench, the result shows that both of the tools are quite efficient and cost less than 2 seconds since these apps are small-size and contain only a few taint flows. For Google-Play and AndroZoo, we plot the analysis times over the code size and memory size in Fig. 6 and Fig. 7. From the graphs, we can observe that FastDroid costs much less time than FlowDroid in general. Furthermore, FastDroid is more stable than FlowDroid and costs less than 50 seconds for most of the apps. The maximum, minimum and average times of analysis are listed in Table 4. We notice that the analysis times are not directly correlated with the code size or memory size, some small-size apps could consume much more time than large-size apps.

Besides, we divide the apps into several groups by code size and memory size and then calculate the average runtimes of analysis on these groups. As Fig. 8 shows, FastDroid takes less time than FlowDroid on all groups. For Google-Play, the average runtimes of both tools increase with the code size and memory size. However, the analysis time of FastDroid increases more slowly and remains at a low cost below 10 seconds when the size increases. In contrast, the analysis time of FlowDroid increases rapidly when the size increases. Specifically, on the group of more than 100K lines of code, the average time of FlowDroid rises to 152.42 seconds, over 22 times longer than the time spent on the group of less than 15K, which is 6.79 seconds. For AndroZoo, the situation is almost the same except for the group of more than 10MB. The analysis times of both tools in this group decrease compared with the group of 5 to 10MB memory size, which indicates that the memory size could not directly determine the analysis time. FastDroid remains at a low cost below 30 seconds on all groups. In summary, from the result we see that FastDroid is more efficient and stable than FlowDroid.

**Table 5 – Test results on DroidBench 2.0.**

| App Name | FLD 1.0 | FLD 2.0 | FastDroid | App name | FLD 1.0 | FLD 2.0 | FastDroid |
|---|---|---|---|---|---|---|---|
| Aliasing | | | | StringFormatter1 | ○ | ○ | ⊛ |
| Merge1 | ⋆ | ⋆ | ⋆ | StringPatternMatching1 | ⊛ | ⊛ | ⊛ |
| AndroidSpecific | | | | StringToCharArray1 | ⊛ | ⊛ | ⊛ |
| ApplicationModeling1 | ○ | ○ | ○ | StringToOutputStream1 | ⊛ | ⊛⋆ | ⊛ |
| DirectLeak1 | ⊛ | ⊛ | ⊛ | UnreachableCode | | | |
| InactiveActivity | | | | VirtualDispatch1 | ⊛ | ⊛⋆ | ⊛ |
| Library2 | ⊛ | ⊛ | ⊛ | VirtualDispatch2 | ⊛⋆ | ⊛⋆ | ⊛⋆ |
| LogNoLeak | | | | VirtualDispatch3 | ⋆ | ⋆ | |
| Obfuscation1 | ⊛ | ⊛ | ⊛ | VirtualDispatch4 | | | |
| Parcel1 | ○ | ○ | ⊛ | ImplicitFlows | | | |
| PrivateDataLeak1 | ⊛ | ⊛ | ○ | ImplicitFlow1 | ○ | ○ | ⊛ |
| PrivateDataLeak2 | ⊛ | ⊛ | ⊛ | ImplicitFlow2 | ○ | ○ | ⊛ |
| PrivateDataLeak3 | ⊛○ | ⊛○ | ⊛○ | ImplicitFlow3 | ○○ | ○○ | ⊛○ |
| PublicAPIField1 | ○ | ○ | ⊛ | ImplicitFlow4 | ○ | ○ | ⊛ |
| PublicAPIField2 | ⊛⋆ | ⊛⋆ | ○ | InterComponentCommunication | | | |
| ArraysAndLists | | | | ActivityCommunication1 | ⊛ | ⊛ | ⊛ |
| ArrayAccess1 | ⋆ | ⋆ | | ActivityCommunication2 | ○ | ⊛ | ⊛ |
| ArrayAccess2 | ⋆ | ⋆ | | ActivityCommunication3 | ○ | ○ | ⊛ |
| ArrayCopy1 | ⊛ | ⊛ | ⊛ | ActivityCommunication4 | ○ | ⊛ | ⊛ |
| ArrayToString1 | ⊛ | ⊛ | ⊛ | ActivityCommunication5 | ○ | ⊛ | ⊛ |
| HashMapAccess1 | ⋆ | ⋆ | | ActivityCommunication6 | ○ | ○ | ○ |
| ListAccess1 | | | ⋆ | ActivityCommunication7 | ○ | ⊛ | ⊛ |
| MultidimensionalArray1 | ⊛ | ⊛ | ⊛ | ActivityCommunication8 | ○ | ⊛ | ⊛ |
| Callbacks | | | | BroadcastTaintAndLeak1 | ○ | ○ | ○ |
| AnonymousClass1 | ⊛○ | ⊛○ | ⊛⊛ | ComponentNotInManifest1 | ⋆ | ⋆ | |
| Button1 | ⊛ | ⊛ | ⊛ | EventOrdering1 | ⊛ | ⊛ | ⊛ |
| Button2 | ⊛⊛⋆⋆ | ⊛⊛⋆⋆ | ⊛⊛ | IntentSink1 | ○ | ○ | ⊛ |
| Button3 | ⊛ | ⊛ | ⊛ | IntentSink2 | ⊛ | ⊛ | ⊛ |
| Button4 | ⊛ | ⊛ | ⊛ | IntentSource1 | ○○ | ⊛⊛ | ⊛⊛ |
| Button5 | ⊛ | ⊛ | ⊛ | ServiceCommunication1 | ○ | ○ | ○ |
| LocationLeak1 | ⊛⊛ | ⊛⊛ | ⊛⊛ | SharedPreferences1 | ⊛ | ⊛ | ⊛ |
| LocationLeak2 | ⊛⊛ | ⊛⊛ | ⊛⊛ | Singletons1 | ○ | ○ | ○ |
| LocationLeak3 | ⊛○ | ⊛○ | ⊛⊛ | UnresolvableIntent1 | ○○ | ⊛⊛ | ⊛⊛ |
| MethodOverride1 | ⊛ | ⊛ | ⊛ | InterAppCommunication | | | |
| MultiHandlers1 | | | | Echoer | ○ | ○ | ⊛ |
| Ordering1 | | | | SendSMS | ⊛○ | ⊛⊛ | ⊛⊛ |
| RegisterGlobal1 | ⊛ | ⊛ | ⊛ | StartActivityForResult1 | ⊛⋆ | ⊛⋆ | ⊛⋆ |
| RegisterGlobal2 | ⊛ | ⊛ | ⊛ | Lifecycle | | | |
| Unregister1 | ⋆ | ⋆ | ⋆ | ActivityLifecycle1 | ⊛ | ⊛ | ⊛ |
| EmulatorDetection | | | | ActivityLifecycle2 | ⊛ | ⊛ | ⊛ |
| ContentProvider1 | ⊛⊛ | ⊛⊛ | ⊛⊛ | ActivityLifecycle3 | ⊛ | ⊛ | ⊛ |
| IMEI1 | ○○ | ○○ | ⊛○ | ActivityLifecycle4 | ⊛ | ⊛ | ⊛ |
| PlayStore1 | ○○ | ○○ | ⊛⊛ | ActivitySavedState1 | ⊛ | ⊛ | ⊛⋆ |
| FieldAndObjectSensitivity | | | | ApplicationLifecycle1 | ⊛ | ⊛ | ⊛ |
| FieldSensitivity1 | | | | ApplicationLifecycle2 | ⊛ | ⊛ | ⊛ |
| FieldSensitivity2 | | | | ApplicationLifecycle3 | ⊛ | ⊛ | ⊛ |
| FieldSensitivity3 | ⊛ | ⊛ | ⊛ | AsynchronousEventOrdering1 | ⊛ | ⊛ | ⊛ |
| FieldSensitivity4 | | | | BroadcastReceiverLifecycle1 | ⊛ | ⊛ | ⊛ |
| InheritedObjects1 | ⊛ | ⊛ | ⊛ | BroadcastReceiverLifecycle2 | ⊛ | ⊛ | ⊛ |
| ObjectSensitivity1 | | | | EventOrdering1 | ⊛ | ⊛ | ⊛ |
| ObjectSensitivity2 | | | | FragmentLifecycle1 | ⊛ | ⊛ | ⊛ |
| GeneralJava | | | | FragmentLifecycle2 | ○ | ○ | ⊛ |
| Clone1 | ⊛ | ⊛ | ⊛ | ServiceLifecycle1 | ⊛ | ⊛ | ⊛ |
| Exceptions1 | ⊛ | ⊛ | ⊛ | ServiceLifecycle2 | ⊛ | ⊛ | ⊛ |
| Exceptions2 | ⊛ | ⊛ | ⊛ | SharedPreferenceChanged1 | ⊛ | ⊛ | ⊛ |
| Exceptions3 | ⋆ | ⋆ | ⋆ | Reflection | | | |
| Exceptions4 | ⊛ | ⊛ | ⊛ | Reflection1 | ⊛ | ⊛ | ⊛ |

(*continued on next page*)

**Table 5 (*continued*)**

| App Name | FLD 1.0 | FLD 2.0 | FastDroid | App name | FLD 1.0 | FLD 2.0 | FastDroid |
|---|---|---|---|---|---|---|---|
| FactoryMethods1 | ⊛⊛ | ⊛⊛ | ⊛⊛ | Reflection2 | ◯ | ◯ | ◯ |
| loop1 | ⊛ | ⊛ | ⊛ | Reflection3 | ◯ | ◯ | ◯ |
| loop2 | ⊛ | ⊛ | ⊛ | Reflection4 | ◯ | ◯ | ◯ |
| Serialization1 | ◯ | ◯ | ⊛ | Threading | | | |
| SourceCodeSpecific1 | ⊛ | ⊛ | ⊛ | AsyncTask1 | ⊛ | ⊛ | ⊛ |
| StartProcessWithSecret1 | ⊛ | ⊛ | ⊛ | Executor1 | ⊛ | ⊛ | ⊛ |
| StaticInitialization1 | ◯ | ◯ | ⊛ | JavaThread1 | ⊛ | ⊛ | ⊛ |
| StaticInitialization2 | ⊛ | ⊛ | ⊛ | JavaThread2 | ⊛ | ⊛ | ◯ |
| StaticInitialization3 | ◯ | ◯ | ◯ | Looper1 | ⊛ | ⊛ | ◯ |
| Sum, Precision and Recall | | | | | | | |
| ⊛=correct warning(higher is better) | 73 | 83 | 97 | | | | |
| ◯ = missed leak(lower is better) | 40 | 30 | 16 | | | | |
| ⋆ =false warning(lower is better) | 13 | 15 | 7 | | | | |
| Precision $p = ⊛/(⊛ + ⋆)$ | 84.9% | 84.7% | 93.3% | | | | |
| Recall $r = ⊛/(⊛ + ◯)$ | 64.6% | 73.5% | 85.8% | | | | |
| F-measure $2pr/(p + r)$ | 0.73 | 0.79 | 0.89 | | | | |

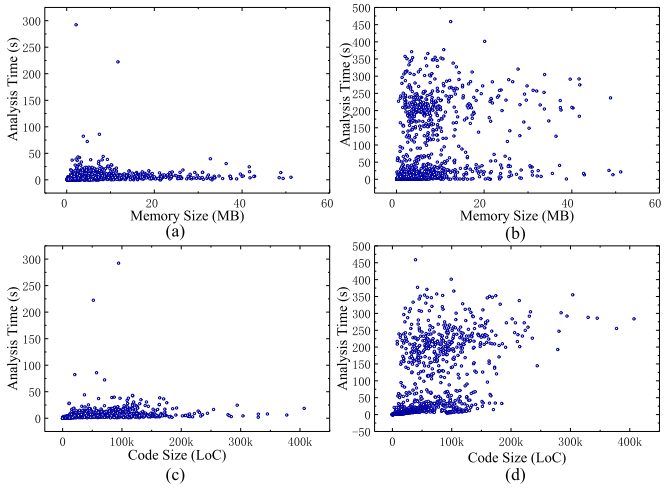


**Fig. 6 – The analysis times of FlowDroid 2.0 and FastDroid on Google-Play vs. code size and memory size. (a) analysis time vs. memory size in FastDroid, (b) analysis time vs. memory size in FlowDroid, (c) analysis time vs. code size in FastDroid and (d) analysis time vs. code size in FlowDroid.**

## 7. Limitation

This work has a few limitations as follows. First, as our experiments show, FastDroid is not able to detect some categories of taint flows, since our tool utilizes the Jimple code and call graph provided by FlowDroid and has the same limitation of FlowDroid. Second, as discussed in Section 5, there exist only a few complicated taint flows in real-world apps that have numerous aliases or pass through a lot of functions. FastDroid may require more than the usual iterations to detect all taint values of these taint flows and then cost considerable analysis time. To obtain a tradeoff between soundness and efficiency, we can set an upper limit for the times of iterations in the tool's configuration.

## 8. Related work

There are several approaches of static taint analysis for Android apps.

At the early stage of research for Android security, several lightweight tools Barrera et al. (2010b); Enck et al. (2009); Felt et al. (2011) were proposed, which are basically based on the analysis of permissions requested or the usage of security-relevant API methods invoked. Kirin Enck et al. (2009) provides a lightweight analysis by using rules of permissions to certificate apps. A. P. Felt et al. Felt et al. (2011) propose a methodology to analyze the Android permission model by employing a matrix-based visualization. D. Barrera et al. Barrera et al. (2010a) search all the API methods called in an app and map those API methods to the permissions for detecting over-privilege. These approaches make a coarse anal-

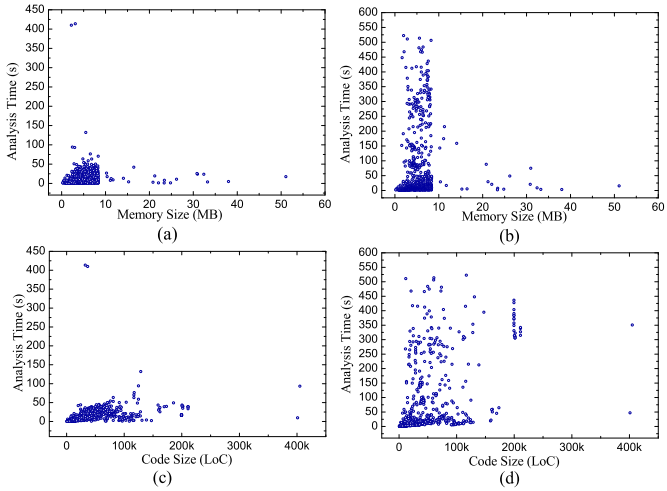


**Fig. 7 – The analysis times of FlowDroid 2.0 and FastDroid on AndroZoo vs. code size and memory size. (a) analysis time vs. memory size in FastDroid, (b) analysis time vs. memory size in FlowDroid, (c) analysis time vs. code size in FastDroid and (d) analysis time vs. code size in FlowDroid.**

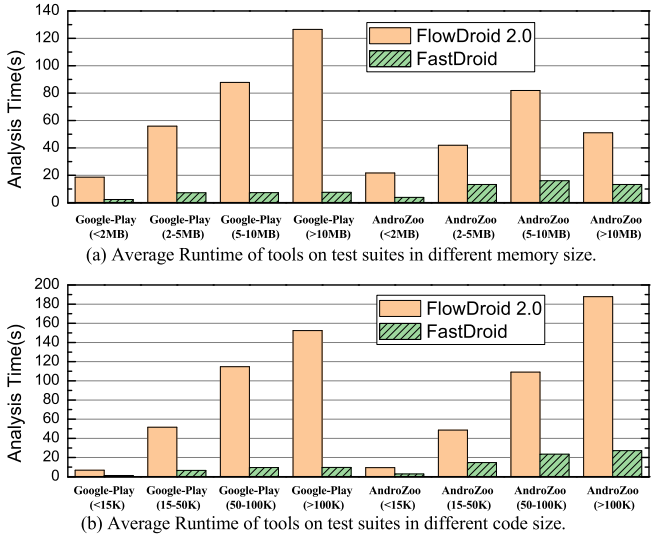


**Fig. 8 – The analysis times of FlowDroid 2.0 and FastDroid on Google-Play and AndroZoo in different sizes. For code size, there are four groups: less than 15K lines, 15-50K lines, 50-100K lines, more than 100K lines; for memory size, there are four groups: less than 2MB, 2-5MB, 5-10MB, more than 10MB.**

ysis for the code, thus have smaller cost but less precision. In contrast, FastDroid performs an in-depth static analysis and reaches a high precision and recall ratio.

FlowDroid Arzt et al. (2014) and IccTA Li et al. (2015) are the most relevant works with our study. FlowDroid is a sophisticated, open-source static taint analysis tool for Android apps that is context, flow, field, and object-sensitive. It uses a dummy main method to model the Android life-cycle and callbacks and performs taint analysis within the CFG based on the IFDS framework which is for solving inter-procedural distributive subset problems. FlowDroid combines the forward-taint analysis with an on-demand backward alias analysis. However, FlowDroid does not support inter-component taint analysis and the cost is quite high especially for the large-scale apps. IccTA integrates FlowDroid with Epicc Octeau et al. (2013) and IC3 Octeau et al. (2015) and could detect inter-component privacy leaks. It solves the ICC problem with an ingenious way, which generates a helper class *IpcSC* and a helper method *redirect* to modify the code related with ICC. However, these operations take more time and make the cost even higher. By contrast, FastDroid designs a novel method for taint analysis, which improves the efficiency significantly.

Apposcopy Feng et al. (2014) is a semantics-based approach for detecting Android malware that can steal private data. It firstly utilizes a high-level language to specify signatures that describe semantic characteristics of different malware families. Then, a static analysis is applied to a given application to decide whether it matches a malware signature or not. The drawback is that Apposcopy cannot be a general tool for taint analysis and not suitable for all kinds of malware especially new variants of malware. The malware families

must have been known and their signatures should be designed previously. DroidSafe Gordon et al. (2015) is another static analysis tool for detection of sensitive information leaks. DroidSafe models the Android runtime completely and precisely and performs an object sensitivity and flow insensitivity information-flow analysis. However, the cost of DroidSafe is even higher than the FlowDroid. Amandroid Wei et al. (2014) provides a general static framework for security analysis of Android apps and supports ICC analysis. To build generic framework for multiple security analyses, Amandroid calculates points-to information of all the objects with additional computing cost. While Amandroid does not support some ICC methods, such as *startActivityForResult* and its runtime of taint analysis is a little higher than FlowDroid, while FastDroid is faster the FlowDroid. Scandroid Fuchs et al. (2009) and LeakMiner Yang and Yang (2012) perform static analysis, but use limited models of Android runtime that have less precision.

There are some fundamental works done by previous approaches for the taint analysis. ICC analysis is a basic step for modeling the Android Apps, such as IC3 Octeau et al. (2015) and Epicc Octeau et al. (2013), which are tools to resolve inter-component communication information in the apps. IC3 provides various ICC information including ICC methods, targets, data fields. In addition, the study on source and sink methods is essential for the taint analysis, SuSi Rasthofer et al. (2014) proposes a novel machine-learning guided approach to identify and categorize the source and sink methods in the Android app with high accuracy.

In recent years, a few approaches for precise taint analysis that improve the efficiency spring up. H. Cai et al. Cai and Jenkins (2018) design an approach that decreases the search space of taint analysis from the entire code to the parts of the code that are different from the previous versions of apps to reduce cost. However, the approach will obviously lose the advantage if the app does not have any earlier version. SparseDroid He et al. (2019) accelerates the taint analysis by sparsifying the traditional IFDS algorithm, and reuses the other FlowDroid's modules related to taint analysis and alias analysis. Compared with SparseDroid, our tool utilizes a novel approach for taint analysis and alias analysis, andalso improves the precision and recall. Manuel Benz et al. Benz et al. (2020) investigate how to incorporate heap snapshots into static analysis and explore the impact of different usages of heap snapshots. The static-fallback approach they proposed has better runtime performance and achieves a higher F-measure compared to the full statical approach. This static-fallback approach is actually a blended analysis, while FastDroid performs a pure static analysis.

## 9. Conclusion

This paper addresses the efficiency problem of detecting sensitive data leaks for Android apps. We present FastDroid, an efficient and precise approach for taint analysis. Unlike the previous approaches, we focus on the propagation of taint values rather than the traditional data flow analysis to improve the efficiency. A serials of experiments show that FastDroid reaches a better comprehensive performance on the precision and recall and improves the efficiency significantly compared with FlowDroid. In our future work, we plan to further improve the precision and recall of FastDroid. To do this, we are going to refine more taint rules according to the feature of various malware.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## CRediT authorship contribution statement

**Jie Zhang:** Conceptualization, Methodology, Software, Writing - original draft. **Cong Tian:** Conceptualization, Investigation, Writing - review & editing. **Zhenhua Duan:** Writing - review & editing, Supervision.

## Acknowledgment

This research is supported by Major Program of the Ministry of Science and Technology of China with Grant No. 2018AAA0103202, NSFC with Grants No. 61732013, and No. 61751207, Key Science and Technology Innovation Team of Shaanxi Province with Grant No. 2019TD-001.

## REFERENCES

Aafer Y, Du W, Yin H. Droidapiminer: Mining api-level features for robust malware detection in android. In: International conference on security and privacy in communication systems. Springer; 2013. p. 86–103.

Allix K, Bissyandé TF, Klein J, Le Traon Y. Androzoo: Collecting millions of android apps for the research community. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR). IEEE; 2016. p. 468–71.

Ammons G, Larus JR. Improving data-flow analysis with path profiles. Acm Sigplan Notices 1998;33(5):72–84.

Arzt S. Static data flow analysis for android applications. Technische Universität; 2017.

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Traon YL, Octeau D, Mcdaniel P. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices 2014;49(6):259–69.

Barrera D, Kayacik HG, van Oorschot PC, Somayaji A. A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the 17th ACM conference on Computer and communications security. ACM; 2010. p. 73–84.

Barrera D, Kayacik HG, Van Oorschot PC, Somayaji A. A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the 17th ACM conference on Computer and communications security. ACM; 2010. p. 73–84.

Bartel A, Klein J, Traon YL, Monperrus M. Dexpler:converting android dalvik bytecode to jimple for static analysis with soot. In: ACM Sigplan International Workshop on State of the Art in Java Program Analysis; 2012. p. 27–38.

Benz M, Kristensen EK, Luo L, Borges Jr P, Bodden E, Zeller A, et al. Heapsnleaks: How heap snapshots improve android taint analysis. ICSE'20: International Conference onSoftware Engineering, 2020.
Bodden E. Inter-procedural data-flow analysis with ifds/ide and soot. In: ACM Sigplan International Workshop on the State of the Art in Java Program Analysis; 2012. p. 3–8.
Cai H, Jenkins J. Leveraging historical versions of android apps for efficient and precise taint analysis. In: Proceedings of the 15th International Conference on Mining Software Repositories. ACM; 2018. p. 265–9.
Enck W, Gilbert P, Han S, Tendulkar V, Chun BG, Cox LP, Jung J, Mcdaniel P, Sheth AN. Taintdroid:an information-flow tracking system for realtime privacy monitoring on smartphones. Commun. ACM 2010;57(3):393–407.
Enck W, Octeau D, McDaniel PD, Chaudhuri S. A study of android application security., 2; 2011. p. 2.
Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on Computer and communications security. ACM; 2009. p. 235–45.
Fastdroid tool, https://github.com/zhangjie-xd/FastDroid.git 2020.
Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security. ACM; 2011. p. 627–38.
Felt AP, Ha E, Egelman S, Haney A, Chin E, Wagner D. Android permissions:user attention, comprehension, and behavior. In: Proceedings of the Eighth Symposium on Usable Privacy and Security; 2012. p. 1–14.
Feng Y, Anand S, Dillig I, Aiken A. Apposcopy: semantics-based detection of android malware through static analysis. In: The ACM Sigsoft International Symposium; 2014. p. 576–87.
Fritz, C., Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2013. Highly precise taint analysis for android applications.
Fuchs, A.P., Chaudhuri, A., Foster, J.S., 2009. Scandroid: Automated security certification of android applications.
Gascon H, Yamaguchi F, Arp D, Rieck K. Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM workshop on Artificial intelligence and security. ACM; 2013. p. 45–54.
Gibler C, Crussell J, Erickson J, Chen H. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In: International Conference on Trust and Trustworthy Computing. Springer; 2012. p. 291–307.
GitHub, 2016. Droidbenchbenchmarks. https://github.com/secure-software-engineering/DroidBench.
GitHub, 2018. Flowdroid 2.0. https://github.com/secure-software-engineering/soot-infoflow-android/wiki.
Glick, K., 2015. Support for 100mb apks on google play. http://android-developers.blogspot.com/2015/09/support-for-100mb-apks-on-google-play.html.
Google, 2019. Intents and intent filters. https://developer.android.google.cn/guide/components/intents-filters.
Gordon MI, Kim D, Perkins J, Gilham L, Nguyen N, Rinard M. In: Network and Distributed System Security Symposium. Information-flow analysis of android applications in droidsafe; 2015.
He D, Li H, Wang L, Meng H, Zheng H, Liu J, Hu S, Li L, Xue J. Performance-boosting sparsification of the ifds algorithm with applications to taint analysis. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2019. p. 267–79.
Huang SS, Green TJ, Loo BT. Datalog and emerging applications: an interactive tutorial. In: ACM SIGMOD International Conference on Management of Data; 2011. p. 1213–16.
Kassner, M., 2012. Google play: Android's bouncer can be pwned. https://www.techrepublic.com/blog/it-security/-google-play-androids-bouncer-can-be-pwned/.
Lam P, Bodden E, Lhoták O, Hendren L. In: Cetus Users and Compiler Infrastructure Workshop. The soot framework for java program analysis: a retrospective; 2011.
Li L, Bartel A, Bissyande TF, Klein J. Iccta: Detecting inter-component privacy leaks in android apps. In: The International Conference on Software Engineering; 2015. p. 280–91.
Li L, Bissyandé TF, Papadakis M, Rasthofer S, Bartel A, Octeau D, Klein J, Traon L. Static analysis of android apps: a systematic literature review. Inf. Softw. Technol. 2017;88:67–95.
Lu L, Li Z, Wu Z, Lee W, Jiang G. Chex: statically vetting android apps for component hijacking vulnerabilities. In: ACM Conference on Computer and Communications Security; 2012. p. 229–40.
Luo L, Bodden E, Späth J. A qualitative analysis of android taint-analysis results. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2019. p. 102–14.
Matney, L., 2018. Google has 2 billion users on android, 500m on google photos. https://techcrunch.com/2017/05/17/google-has-2-billion-users-on-android-500m-on-google-photos/.
Octeau D, Luchaup D, Dering M, Jha S, Mcdaniel P. Composite constant propagation: application to android inter-component communication analysis. In: Ieee/acm IEEE International Conference on Software Engineering; 2015. p. 77–88.
Octeau D, Mcdaniel P, Jha S, Bartel A, Bodden E, Klein J, Traon YL. Effective inter-component communication mapping in android with epicc: an essential step towards holistic security analysis. In: Usenix Conference on Security; 2013. p. 543–58.
Pauck F, Bodden E, Wehrheim H. Do android taint analysis tools keep their promises?. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2018. p. 331–41.
PETROVAN, B., 2015. Google is now manually reviewing apps that are submitted to the play store! https://www.androidauthority.com/google-now-manually-reviewing-apps-submitted-to-play-store-594879/.
Project, A.O.S., 2019. Permissions overview. https://developer.android.com/guide/topics/permissions/overview.
Rasthofer S, Arzt S, Bodden E. In: Network and Distributed System Security Symposium. A machine-learning approach for classifying and categorizing android sources and sinks; 2014.
Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM; 1995. p. 49–61.
Statcounte, 2020. Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide.
Vallee-Rai, R., Hendren., L.J., 1998. Jimple: Simplifying java bytecode for analyses and transformations, 18–28.
Wei F, Roy S, Ou X, Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In: ACM Sigsac Conference on Computer and Communications Security; 2014. p. 1329–41.
Wei F, Roy S, Ou X, et al. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Transactions on Privacy and Security (TOPS) 2018;21(3):14.
Yang W, Xiao X, Andow B, Li S, Xie T, Enck W. Appcontext:differentiating malicious and benign mobile app behaviors using context. In: Ieee/acm IEEE International Conference on Software Engineering; 2015. p. 303–13.
Yang Z, Yang M. Leakminer: Detect information leakage on android with static taint analysis. In: Software Engineering; 2012. p. 101–4.
Zhang J, Tian C, Duan Z. Fastdroid: efficient taint analysis for android applications. In: Proceedings of the 41st International

Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019. IEEE / ACM; 2019. p. 236–7. doi:10.1109/ICSE-Companion.2019.00092.

**Jie Zhang** is a lecturer at Xidian University and is currently pursuing the Ph.D. degree with Xidian University. He received the B.S. and M.S. degrees in computer science and technology from Xidian University, Xi'an, P.R. China, in 2008 and 2011, respectively. His research interests include mobile security, malware analysis, and privacy.

**Cong Tian** received the BS, MS, and Ph.D. degrees in computer science from Xidian University, Xi'an, P.R. China, in 2004, 2007, and 2009, respectively. She is currently a professor in the Institute of Computing Theory and Technology (ICTT), Xidian University, Xian, P.R. China. She was a visiting postdoctoral researcher in Hosei University, Japan, from 2010 to 2011. Her research interests include theories in model checking, software analysis, formal verification of software systems, and software engineering.

**Zhenhua Duan** received the Ph.D. degrees in computer science from both Newcastle University and the University of Sheffield, United Kingdom. He is currently a professor in the Institute of Computing Theory and Technology (ICTT), Xidian University, Xi'an, P.R. China. His research interests include model checking, temporal logics, formal verification of software systems, and temporal logic programming.