A Component-Sensitive Static Analysis Based Approach for Modeling Intents in Android Apps

Negarsadat Abolhassani
University of Southern California
USA

William G.J. Halfond
University of Southern California
USA

***Abstract*—The Android Inter Component Communication (ICC) model plays an important role in providing users with a wide range of features both within and across apps. Accurate information about ICCs is important for a range of program analysis-based security and verification techniques. These techniques use static analysis to infer links between Android components in bundles of apps and then identify potential security problems with these communications. However, existing ICC analyses have limitations in terms of their accuracy when Intents are constructed using ICC information. To address these limitations, we introduce a multi-level component-sensitive static analysis technique to efficiently and accurately compute ICC information in these scenarios. We compared our approach with state of the art ICC analysis techniques and found that our approach is more accurate and has a faster execution time.**

***Index Terms*—program analysis, Android apps, Inter component communication, Intent**

I. Introduction

The Android Inter Component Communication (ICC) model has become popular and widely used by developers. This mechanism enables the reusability of information and services within and across apps, and mainly happens through passing asynchronous messages called Intents [1]. Due to the important and widespread role ICC model plays in modern Android apps, verifying the correctness and safety of ICC usage has become increasingly important and an active area of research in the community. This has included techniques for understanding the Graphical User Interface (GUI) structure of apps (e.g., [2], [3], [4], [5]), fault localization (e.g., [6], [7]) and the discovery of vulnerabilities that can be caused or exploited by ICC, (e.g., [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]).

All of these techniques require the ability to identify and model apps' ICC information. Specifically the names and values of attributes that define an Intent message. Accurately identifying this information is both essential and challenging. It is essential because attributes defined in Intents are used by the Android OS to route Intents to specific recipients and to carry data between components. Higher-level analyses, such as vulnerability detection, use this information to track targets and recipients of Intents and to understand the kind of data that is being communicated (e.g., malicious). However, several such analyses have specifically attributed inaccuracies in the underlying ICC analysis as the reason for their own inaccuracies (e.g., [18], [7]). The main reason for these inaccuracies is that identifying Intent information is very challenging. Modern Android apps do not have to declare a priori the attributes they might contain nor the values the attributes are set to, meaning that both the target of an Intent and its content may not be known until runtime. This situation happens since, as we show in Section II, Intents can be built and modified via operations along the various paths in an app's implementation. These operations can add or set attributes of Intents based on the internal logic of the containing app. Furthermore, as we also show in Section II, the values of attributes and in some cases, their names as well, can be defined by values supplied by other Intents originating from components in the app or other apps. Altogether these aspects make the identification of Intent information very challenging.

The importance of identifying Intent-based communication has led the research community to develop several different techniques for this purpose. One well-known technique for identifying the content of Intents is IC3 [19], [20]. This technique employs static analysis techniques to identify the names and values of Intent attributes. However, as we discuss in more depth in Section II, IC3 is not able to accurately model the value of Intent attributes when they are built using information received by components. Therefore, when an Intent's content depends on ICC information, IC3 is unable to accurately account for this information. Many other techniques (e.g., StoryDroid [3] and RAICC [21]), which are built on IC3 to identify Intent information, also inherit these limitations. Another well-known static analysis-based technique for ICC analysis is ICCBot [22]. It has limitations similar to IC3 and does not model Intents that use the information received from other Intents. Dynamic analyses (e.g., [23], [24], [25]) are also used for identifying Intent information that can be sent by apps or performing Intent matching. However, the completeness of dynamic analysis techniques depends on how thoroughly the apps are executed at runtime, and it can be challenging to ensure that all execution paths in an app are executed. For example, some paths may only execute under hard to trigger conditions.

The limitation of existing techniques directly motivates the design of our approach for modeling Intents in Android apps. At a high-level our approach takes a bundle of apps as input and performs a static analysis on the apps in the bundle, producing a graph that shows which Android components in the bundle can communicate with each other and the content of the Intents the components use to communicate.

The design of our analysis makes two important contributions over state of the art. First, we introduced a novel concept called *component-sensitivity*, which allows an ICC analysis to accurately model ICC objects when they are built from ICC information received by their component. Second, we designed a two-level ICC analysis that is component sensitive for accurately computing ICCs in a bundle of apps. We implemented and compared our approach against two state of the approaches for ICC analysis. Our results showed that the component-sensitive analysis enabled our approach to more accurately and completely model Intents and ICCs in bundles of apps. Our analysis was also fast, accomplishing these results in a similar or lesser amount of time than the other approaches. These results indicate that our approach could serve a practical and important role in providing accurate Intent information for a range of ICC-related verification tasks.

Our paper is organized as follows. In Section II, we provide background information about the Android communication mechanism and motivate our bundle analysis approach using an example. Our approach is presented in Section III and its evaluation is discussed in Section IV. We discuss related work in Section VI and conclude in Section VII.

II. Background and Motivating Example

Android's architecture enables apps to share information and services among each other. This is facilitated through the Inter Component Communication (ICC) model mechanism that allows the building blocks of Android apps, which are called components, to communicate with each other. ICCs can happen within individual apps or across them in *bundles of apps*, which are sets of apps running together in the same environment (e.g., the set of apps on a user's device). ICC mainly happens through passing asynchronous messages called Intents [1] between components of types Activity, Broadcast Receiver, or Service [26].

Intent Attributes. Android defines a set of attributes for Intents: (1) `COMPONENT` and `PACKAGE` - specify the name and the location of the component that should receive the Intent; (2) `ACTION` - the type of action the receiver component should perform, (3) `CATEGORY` – a description of the component that should receive the Intent, (4) `TYPE`, and `DATA` - description of the information to be acted on; (5) `FLAG` that provide instructions for the Android system about how to launch components, and (6) `EXTRA` that is itself a collection of additional name-value pairs, which may be defined by the sending component. We refer to Intent attributes except for `EXTRA` as the *standard Intent attributes* and the additional name-value pairs introduced by `EXTRA` as nonstandard Intent attributes.

Intent Routing. The Android operating system is responsible for performing ICC by routing Intents between components [27]. The routing decision is based on (1) the semantics of the expected ICC (e.g., returning a result to the component requesting it), (2) the content of Intents, and (3) the components' declarations of their characteristics (i.e., *Intent Filters* [28]). If an Intent provides the full target component name (i.e., *explicit Intent*); the Android OS routes it directly to the specified component. If an Intent only specifies the characteristics of the target component (i.e., *Implicit Intent*); the Android OS routes it by matching the values of the Intent's attributes against the Intent Filters declared by each app to determine if one of the app's component can accept the Intent. Also, upon identifying the target component, depending on the semantics of the ICCs, Android OS determines the exact code location receiving an Intent. For example, the Intent starting a Service component would be received by the `onStartCommand` callback and a result would be sent to the `onActivityResult` callback of the requester Activity.

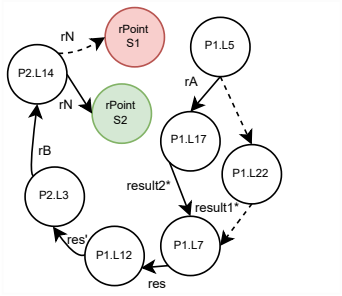


Fig. 1: iGraph of the motivating example. Each node represents a code point that sends or receives an Intent. Edges are labeled with the Intents flowing between these code points. The solid lines represent correct Intent flows and the dashed lines are incorrect.

Motivating Example. To illustrate and motivate our approach, we explain the ICCs among the three Components shown in Program 1 and Program 2. In Program 1, Activity Component $A1$, creates an Intent, rA, sets its destination to Activity Component $A2$, and sends it at Lines 3 to 5 of Program 1. $A1$ starts $A2$ with rA to request information and $A2$ responds to this request with an Intent at Line 17 after putting a new key-value pair $\langle$"id", 2$\rangle$ in the `EXTRA` of the Intent. The Android OS delivers the result to $A1$ at Line 7. $A1$ reuses the result Intent, res, by updating its `ACTION` attribute and adding two more name-value pairs to its `EXTRA` at Lines 8, 10 and 11. Then, res, which has the following content, { $\langle$`ACTION:{"B_ACTION"}`$\rangle$, $\langle \epsilon_{\texttt{url}}$`:{getUrl()}`$\rangle$, $\langle \epsilon_{\texttt{class}}$`:{"com.example.app.S"}`$\rangle$, $\langle \epsilon_{\texttt{id}}$`:{"2"}`$\rangle$, } is sent out as a broadcast at Line 12. In our notation, we use ϵ to denote that a name-value pair is defined as part of the `EXTRA` attribute collection. Assume that B has declared the appropriate Intent Filter (shown at Line 1 of Program 2) to receive res as rB at Line 3 of Program 2. B uses the content of rB to construct a new Intent, rN, and starts a Service. To do so, it extracts information from the `EXTRA` attribute of rB at Lines 5 to 7 and 12 and sets the destination of rN at Line 8, after applying the String operations `concat` and `substring` on the extracted value. Now, the values of `PACKAGE` and `COMPONENT` in rN should be `"com.example.app"` and `"com.example.app.S2"`, respectively. B passes rB and rN to the $updateData$ method to set the `DATA` attribute of rN and start a service with it. The `DATA` is derived from the value of "url" in the `EXTRA` of rB (Lines 12 and 13

of Program 2). Eventually, $param2$ at Line 14 would start Service $S2$ while carrying the return value of `getURL` defined at Line 9 of Program 1 as its `DATA` attribute. Figure 1 shows the Intent flows. In this figure, nodes are labeled as $P\#.L\#$, where $P\#$ refers to Program 1 or Program 2 and $L\#$ denotes the line number in that program. The "result*" stands for the result Intent at `setResult` invocations and "res'" indicates that Intent res has been modified.

Challenges. Even the seemingly simple example shown in Program 1 and Program 2 would cause problems for the current state of the art approaches for computing Intents (e.g., IC3 [20], [19]).

The problem is with modeling Intents when they are constructed from **received Intent**'s information. This construction of Intents can happen by repurposing a received Intent, by modifying its Intent attribute (e.g., res in Line 8 of Program 1) or extracting information from it to set another Intent's Intent attributes (e.g., rN is constructed from the content of rB in Program 2). We refer to such dependencies as *Inter Component Dependency (ICD)*. Current Intent analysis techniques are unable to correctly compute the values of attributes defined in this way. Instead, they over-approximate the value as any value or ignore such Intents and attributes all together.

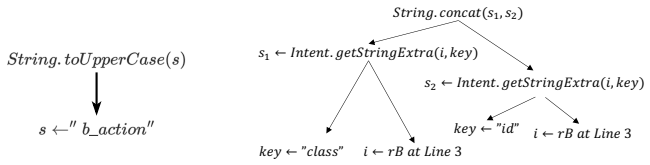
This problem can reduce the accuracy of Intent analyses and negatively impact higher-level analyses that depend on Intent identification. The impact of the limitation mentioned above can be seen even in our small example. Figure 1 shows the graph of the ICC among the components of our example. The solid lines show actual Intent flows among them. The dotted lines represent incorrect flows that would be inferred based on the over-approximations instead of accurately propagating information and including them into the computations. The outcome of such inaccuracies would be identifying an Intent flow to service "S1", which is a false positive. Depending on the context, this false positive ICC might impact other analyses. For example, a data leakage might be falsely detected, if the `DATA` contains sensitive information.

```
1  public class A1 extends Activity {
2      void onStart() {
3          Intent rA = new Intent();
4          rA.setClassName("com.example.app", "com.example.app
               .A2");
5          startActivityForResult(rA);;
6      }
7      void onActivityResult(Intent res) {
8          res.setAction("b_action".toUpperCase());
9          String url = getURL();
10         res.putExtra("url", url);
11         res.putExtra("class", "com.example.app.S");
12         sendBroadcast(res);
13     }
14 }
15 public class A2 extends Activity{
16     void onStart() {
17         setResult(1, new Intent().putExtra("id", "2"));
18     }
19 }
20 public class A3 extends Activity{
21     void onCreate(){
22         setResult(1, new Intent().putExtra("id", "1"));
23     }
24 }
```

Program 1: Activities

```
1  //declares <intent-filter><action android:name="B_ACTION"/></intent
       -filter>
2  public class B extends BroadcastReceiver {
3      void onReceive (Intent rB) {
4          Intent rN = new Intent();
5          String id = rB.getStringExtra("id");
6          String pkg = rB.getStringExtra("class").substring(0, 11)
7          String cmp = rB.getStringExtra("class").concat(id);
8          rN.setClassName(pkg, cmp)
9          dataUpdate(rB, rN)
10     }
11     void dataUpdate(Intent param1, Intent param2){
12         Uri u = Uri.parse(param1.getStringExtra("url")
13         param2.setData(u);
14         startService(param2)
15     }
16 }
```

Program 2: Broadcast Receiver



(a) The model representing the value of the Intent attribute `ACTION` (Line 8 in Program 1)

(b) The model representing cmp which sets the value of Intent attribute `COMPONENT` (Line 7 in Program 2)

Fig. 2: Examples of models representing the value of Intent attributes

III. Component-Sensitive ICC Analysis

The goal of our approach is to accurately and efficiently identify Intent-based ICCs in a bundle of Android apps. As illustrated in Section II, an important aspect of doing this accurately is to account for parts of Intents that are built using information from other Intents. Existing approaches to ICCs analysis either under or over-approximate these parts of the Intents, leading to inaccuracy in the identified ICC. To address this limitation, we introduce a novel concept called *component-sensitivity*, which allows our ICC analysis to identify information that flows into a receiving component from another component via an Intent and accurately model the use of that information in Intents created by the receiving component. As we show in Section IV, component-sensitivity enables our approach to more accurately identify Intent based ICC. Our approach for performing a component-sensitive ICC analysis is structured into two levels. The first level of the analysis is an *intra*-component analysis, which models the Intents that can be generated by each component in the bundle, leaving placeholders for part of the Intents that will later be supplied by Intents from other components (Section III-A). The second level of our analysis is an *inter*-component analysis, which propagates Intent information among the components in the bundle, and uses this information to fill in the placeholders left by the intra-component analysis and compute the Intent-based ICCs (Section III-B).

Algorithm 1: Intra Component Analysis

Input: a: An Android app
Output: $\mathcal{T}$: Reached Intent types at each node in a

1 $\mathcal{T} \leftarrow \emptyset$
2 **foreach** *method* $m \in$*methods of* a *in reverse topological order* **do**
3 **foreach** *statement that defines or modifies an Intent,* $s, \in m$ **do**
4 **if** $s \equiv r =$ *new intent reference* **then**
5 $i \leftarrow \{\langle a, model(s, a)\rangle | a \in$ standard Intent attrs or a $= \epsilon\}\}$
6 **if** $s \equiv r.modify(Attr, V)$ *where* r *is an Intent* **then**
7 **if** *Attr is from* `EXTRA` **then**
8 $M_k \leftarrow model(s, "key")$
9 $M_v \leftarrow model(s, "value")$
10 $i \leftarrow \{\langle \epsilon_a, M_v\rangle | a \in M_k\}$
11 **else**
12 $i \leftarrow \{\langle a, model(s, a)\rangle | a \in Attr\}$
13 $Gen[s] \leftarrow \{\langle r, i\rangle\}$
14 $Kill[s] \leftarrow \{\langle r, i'\rangle |$any $i'\}$
15 $worklist \leftarrow m.entry$
16 **while** $worklist \neq \emptyset$ **do**
17 $s \leftarrow$ remove fMst element of $worklist$
18 $In[s] \leftarrow \bigcup_{p \in preds(s)} Out[p]$
19 $Out' \leftarrow Out[s]$
20 $Q \leftarrow \emptyset$
21 **if** $s \equiv r =$ *new intent reference* **then**
22 $Q \leftarrow Gen[s]$
23 **if** $s \equiv r.modify(Attr, V)$ **then**
24 $R \leftarrow \{\langle r', i\rangle | \langle r', i\rangle \in In[s] \wedge$ r' points to r$\}$
25 **foreach** $\langle r, i\rangle \in R$ **do**
26 $Q \leftarrow Q \cup update(i, Gen[s], s)$
27 **if** s *invokes* $f(.*)$ *or* $r = f(.*)$ **then**
28 $Q \leftarrow substitute(summary[f], In[s])$
29 $Out[s] \leftarrow (In[s] - Kill[s]) \cup Q$
30 **if** $Out' \neq Out[s]$ **then**
31 $worklist \leftarrow worklist \cup$ successors of s
32 $Summary[m] \leftarrow Out[m.exit]$

A. Intra-Component Analysis

The goal of the intra-component analysis is to compute models of the Intents that can be sent by each component in the App Under Analysis (AUA), considering information available only within the component. Although our intra-component analysis shares the same goal as related work (e.g., [19] [22]), these approaches are not component-sensitive, which means their results must either under or over-approximate parts of Intents that are constructed using Intents from other components. Our key insight for addressing this limitation is to design an intra-component analysis that computes models of the Intent creation in each component with placeholders for parts defined from other components, and delays the interpretation (concretization) of these models until more information becomes available via the inter-component analysis. In contrast, related approaches only identify and compute Intent information based on constant Strings, essentially computing and concretizing at the same time, without including information coming from other components (i.e., without component-sensitivity.)

Our intra-component analysis is shown in Algorithm 1. This method is called for each Android app. Our analysis performs a fixed point analysis on the methods of the app, using method summarization to make the analysis inter-procedural and context-sensitive. The methods are analyzed in reverse topological order with respect to the call graph to ensure that a method's summary is computed before any method that calls it is analyzed. Our analysis computes a summary of the Intent Types (iTypes) each method could generate at runtime, where each iType is a set of Intent attribute name-value pairs. Based on our insight described above, the value of each attribute's name and each attribute's value are represented as models with placeholders for content originating from other Intents. (We explain this model in more detail below.) The analysis for each method has two general phases. First, it identifies the Intent related semantics of each statement in a method and computes a model of it's possible values (Line 3 - Line 14). Second, it propagates and combines these models along the paths of the method's Control Flow Graph (CFG) (Line 15 - Line 31) to generate a model that represents the iTypes generated along the various paths.

The first phase of our intra-component analysis (Line 4 - Line 14) plays an integral part in capturing the semantics of Intents and achieving component-sensitivity. During this phase, our approach models the Intent information that can be defined by each statement s in a method and uses this to initialize $Gen[s]$, the Intent information that is propagated in the second phase of the intra-component analysis. For example, consider Line 8 of Program 2. Our approach creates a model of the possible values for cmp, which sets the Intent attribute `COMPONENT`. The model generated by our approach is shown in Figure 2b. This is an Abstract Syntax Tree (AST) like model, where the leaf nodes are constants, variable references, or placeholders for values supplied by other Intents, and the internal nodes represent operations. Given that most Intent attributes are based on Strings [19], we implemented this modeling as a component-sensitive String analysis. We extended an existing general purpose String analysis tool (Violist [29], see Section IV for implementation details) to handle Strings originating from Intent related APIs. Specifically, this extension models such Strings with placeholders, such as the two seen in Figure 2b that represent the "class" and "id" attributes of an Intent used to define the `COMPONENT` attribute at line Line 8 of Program 1. The models with placeholders are key to achieving component-sensitivity since these placeholders represent parts of iTypes that can be resolved via the inter-component analysis. The first phase uses this modeling, which we refer to as *model* in Algorithm 1, to initialize the Gen and $Kill$ sets for two categories of statements.

The **first** category of statements are those that define a new Intent reference. For example, via the `new Intent()` operation or supplied as a parameter to the containing method. Our approach handles these Intents by creating a new iType with a name-value pair for each of the standard Intent attributes and a special name-value pair denoting non-standard attributes (ϵ). For each of these name-value pairs, our approach generates a model of its value. In the case of Intents created directly, the

model represents the default (empty) String value, for Intents passed into the method, the model contains a placeholder that identifies the Intent parameter. The reference to the parameter facilitates resolving the placeholder in the inter-component phase.

The **second** category of statements are those that refine an iType via operations that imply the name and/or value of its attributes. For example, Line 13 of Program 2 refines an iType since it implies that the value of `DATA` attribute will be set to a new value. We obtained a list of all such statements through a one-time analysis of the Android Intent APIs. For simplicity of the algorithm presentation, we use a normalized representation, `Intent.modify(Attr, V)`, for all such statements, where $Attr$ is the set of attribute names being modified and V is their corresponding values. All attribute-modifying APIs defined by the Android SDK can be normalized to this form. For example, `Intent.setData("value")` is represented as: `Intent.modify(DATA, "value")` and `Intent.putExtra(key, value)` can be represented as `Intent.modify("`ϵ_{key}`", "value")`, where ϵ is a special symbol denoting the attribute is an `EXTRA` attribute. For an attribute-modifying statement s that changes a standard Intent attribute, our approach sets the generated iType information to be the name-value pair implied by s (Line 12). Note that only the value needs to be modeled since the name of the API function invoked by s indicates the attribute name (e.g., `setClassname` at Line 8 of Program 2 indicates that the `COMPONENT` and `PACKAGE` attributes will be set.) If s modifies an attribute stored in the extra field (e.g., by calling `Intent.putExtra("key", "value")`), then the process is similar, but both the name and value of the attribute must be modeled since both can be set programmatically (Line 8 - Line 10).

To illustrate the first phase's modeling of the content of Intents, consider the following two examples. First, consider Line 7 in Program 1, which defines a received Intent, res. Here, our approach defines a new iType that contains a name-value pair for each possible Intent attributes. Hence, the Gen set for the first example would be $\{\langle res, i\rangle\}$, where i is the set of name-value pairs of form $\langle a : \{\text{M}_{\text{Line 7}}\}\rangle$, where a is either one of the standard Intent attributes or ϵ. We use the notation $\text{M}_{\text{Line 7}}$ to indicate that the value of each of res's Intent attributes is set based on the Intent received at Line 7. For the second example, consider Line 7 of Program 2. The model computed for the `cmp` variable is shown in Figure 2b. The root of the tree shows a String expression with *concat* as its operation and two arguments, which are themselves the result of two calls to *getStringExtra*. One argument to *getStringExtra* is the Intent received, rB, at Line 3, and the second argument is a key of an `EXTRA` pair. These extras' key names are defined by constant Strings ("id" and "class").

The second phase of the intra-component analysis (Line 15 - Line 31) combines and propagates the iType information that was initialized in the first phase. To do this, our analysis defines $In[s]$ to represent iType information that flows into s from its predecessors in the CFG, and $Out[s]$ to represent the iType information that will flow to the successors of s in the CFG. Our approach computes $Out[s]$ by applying the iType creation semantics of s (i.e., $Gen[s]$) to the incoming iType information (i.e., $In[s]$). The result of this application is represented by Q. Broadly, there are three categories of statements that can determine the value of Q and are shown in our algorithm. The **first** category of statements defines Intent references (Line 21). For these statements, our algorithm directly adds the newly defined iType information in Gen[s] to Q (Line 21 - Line 22). The **second** category of statements modify or set Intent attributes (Line 23). For these, our algorithm updates all incoming iTypes with the same reference as r and updates any of their attributes that match the new name-value pair in Gen[s], before adding the resulting updated iTypes to Q (Line 23 - Line 26). The **third** category of statements invokes functions (Line 27) that themselves modify iTypes. Upon seeing these statements, our approach substitutes iTypes information reaching s (i.e., $In[s]$) into the summary of the invoked method and then propagates the resulting iType information. In our algorithm, this functionality is represented as *substitute*.

The *substitute* represents the concretization of a method summary. For a statement s that invokes a summarized method m, this function takes m's summary as an argument and then substitutes iType information that flows to s via the $In[s]$ set into the summary. To illustrate the application of the function $substitute$, consider Line 9 of Program 2, where rN and rB are passed to the `dataUpdate` method. Our algorithm tracks this by combining the summary of `dataUpdate` with the In set information flowing into this line. The In set at this line contains the iType representing rN: $\{\langle \texttt{PACKAGE} : M_{Line\ 6}\rangle, \langle \texttt{COMPONENT} : M_{Line\ 7}\rangle\}$. In the summary of `dataUpdate`, the iType representing the $param2$ should update the In set information. As the iType in the summary is $\langle \texttt{DATA} : \{M_{Line\ 12}\}\rangle$, the Out set of Line 9 would contain the following iType for rN: $\{\langle \texttt{PACKAGE} : M_{Line\ 6}\rangle, \langle \texttt{COMPONENT}, M_{Line\ 7}\rangle, \langle \texttt{DATA} : M_{Line\ 12}\rangle\}$.

The analysis of a given method (Line 15 - Line 31) terminates when the Out sets of all statements reach a fixed point, meaning that all iTypes have been updated and no further changes are possible. After termination, the Intent-related information propagated to the exit point of method m is used as its summary, which in turn is used wherever m is called from within a. Termination happens since there is a finite upper bound on the content of the Out sets, which is the set of all Intent references and attributes defined in the method, and each iteration of the algorithm causes the Out set to grow monotonically. Referring back to our example, eventually, this phase generates the following summary for method $dataUpdate$: $\{\langle param1, i1\rangle, \langle param2, i2\rangle\}$, where $i1$ is the set of name-value pairs of form $\langle a : \{\text{M}_{\text{Line 3}}\}\rangle$ with a being either one of the standard Intent attributes or ϵ, and $i2$ is $\{\langle \texttt{PACKAGE} : M_{Line\ 6}\rangle, \langle \texttt{COMPONENT}, M_{Line\ 7}\rangle, \langle \texttt{DATA} : M_{Line\ 12}\rangle\}$.

Algorithm 2: Inter Component Analysis

Input: B: an app bundle, S: B's sending points, R: B's receiving points, $\mathcal{T}_S$: Intent types at S
Output: G: iGraph to show Intent flows in B

1 $G.V \leftarrow S \cup R$
2 $\forall v \in G.V \quad (Gen[v], In[v], Out[v] \leftarrow \emptyset)$
3 **foreach** $s \in S$ **do**
4 $\quad Gen[s] \leftarrow \{\langle t, c\rangle | c \in interpret(t, \emptyset) \wedge \langle s, t\rangle \in \mathcal{T}_S\}$
5 $\quad G.E \leftarrow G.E \cup \{\langle s, r, c\rangle | ifICC(s, r, c) \wedge \langle t, c\rangle \in Gen[s] \wedge r \in R\}$
6 $G.V \leftarrow G.V \cup s_A$
7 $I_A \leftarrow$ {Intent launching app|app $\in B$} $\cup$ {Android protected Intents}
8 $G.E \leftarrow G.E \cup \{\langle s_A, r, c\rangle | ifICC(s_A, r, c) \wedge r \in R \wedge c \in I_A\}$
9 $worklist \leftarrow G.V$
10 **while** $worklist \neq \emptyset$ **do**
11 $\quad w \leftarrow$ remove first element of $worklist$
12 $\quad In[w] \leftarrow \bigcup_{p \in preds(w)} Out[p]$
13 $\quad Out' \leftarrow Out[w]$
14 $\quad Out[w] \leftarrow \emptyset$
15 $\quad$ **if** $w \in S$ **then**
16 $\quad\quad$ **foreach** $t \in \{t | \langle t, c_t\rangle \in Gen[w]\}$ **do**
17 $\quad\quad\quad C' \leftarrow interpret(t, In[w])$
18 $\quad\quad\quad Out[w] \leftarrow Out[w] \cup \{\langle t, c'\rangle | c' \in C'\}$
19 $\quad\quad E' \leftarrow \{\langle w, r, c\rangle | ifICC(w, r, c) \wedge r \in R \wedge \langle t, c\rangle \in Out[w]\}$
20 $\quad$ **if** $w \in R$ **then**
21 $\quad\quad Out[w] \leftarrow In[w]$
22 $\quad\quad E' \leftarrow \{\langle w, s, c\rangle | ifIccDep(w, s) \wedge s \in S \wedge \langle t, c\rangle \in Out[w]\}$
23 $\quad G.E[w] \leftarrow G.E[w] \cup E'$
24 $\quad$ **if** $Out' \neq Out[w]$ **then**
25 $\quad\quad Worklist \leftarrow Worklist \cup$ successors of w

B. Inter-Component Analysis

The goal of the inter-component analysis is to compute models of the Intents that can be sent within the Bundle of Android Apps Under Analysis (BUA), taking into account information provided by other components that can be used to resolve placeholders in the models produced by the intra-component analysis. The key challenge in this analysis is to propagate Intent information among components, even when information about their attributes and the target component is only partially known. For example, the target of an Intent sent by one component may be defined by an attribute defined in an Intent received by the component. To address this challenge, we define a fixed point analysis that iterates over an Intent Flow Graph (iGraph), propagating information about the Intents and building edges in the graph as more information becomes available, until no new edges are identified. The iGraph built by our analysis is a labeled directed graph $G = (V, E)$, where V is the set of Intent communication points (i.e., Intent Receiving Program Points (rPoints) and Intent Sending Program Points (sPoints)) in the BUA, and E is a set of labeled directed edges representing Intent flows between these points. Each of these edges is labeled with the content of the Intent (i.e., iType) that may flow from the source node to the target node.

The construction of the iGraph is the primary focus of the inter-component analysis. The key challenge, as described above, is to construct its edges when only partial information about the iTypes is known. Our insight to address this challenge is to do the construction iteratively. First, our approach identifies all iTypes that do not contain any placeholders in their attributes. These iTypes bootstrap the edge definitions in the graph since their content can be resolved precisely without additional information from other components. Next, our approach propagates information about the Intents on these edges to the relevant rPoints of the target component. Our analysis substitutes the Intent information received by a component into any applicable placeholders in the iTypes sent by the component at its sPoints. In turn, any iTypes created by an sPoint that are now fully resolved are also propagated to the sPoint's target component. This process is repeated until no new edges are resolved or added to the graph.

A formal representation of our approach is shown in Algorithm 2. As with our intra-component analysis, we define $Gen[v]$, $In[v]$, and $Out[v]$ functions. However, in this analysis, v is a node in the iGraph, i.e., either an sPoints or rPoints, and each of these functions maps to a set of tuples of the form $\langle t, c\rangle$, where t is an iType, and c is an interpretation of t. Note that the nodes of the iGraph can be completely and accurately identified ahead of time even though the edges among them are not known. Algorithm 2 has two phases, which we explain below.

The first phase of Algorithm 2 (Line 1 - Line 8) initializes the iGraph and sets up the Gen function needed for the fixed point analysis. To do this, our approach iterates over each sPoint, s, interprets the models defining each iType sent at s (Line 4), and assigns these models and their interpretations to $Gen[s]$. The interpretation happens via the *interpret* function, which takes all Intent-related information that arrives at w (i.e., In[w]), which is empty at Line 4, and substitutes it into the relevant placeholders in the Intent type model (t) to produce the set of intent types updated at w. Then our approach constructs an edge for each interpretation of iType in $Gen[s]$ for which it is possible to identify its target component's rPoint r (Line 5). Here, we use `ifICC` to represent an implementation of the standard Android Intent resolution policies for implicit and explicit Intents [27]. Our approach also adds Android-specific system ICCs to the iGraph (Lines 7 and 8). These are important to include since, in our experience, they trigger additional ICCs in apps in a bundle, and the values of attributes received from the system themselves might be passed to other components via Intents. The Android system can send two types of Intents to a component: (1) Launch Intents – an Intent that starts the main Activity of an app (e.g., when the app's icon is clicked); and (2) Protected Intents – Intents to communicate system information, such as a low-battery warning. To include these in the iGraph, our approach represents the Android system as a special sPoint, s_A, and adds this to the graph's node set (Line 6). Since the system Intents' attributes are defined a priori by the Android system documentation, our approach directly creates iTypes to represent them (Line 7). Then our approach adds an edge from s_A to each possible rPoint for each of these iTypes (Line 8).

The second phase of Algorithm 2 iterates over the nodes

in the iGraph, propagating and resolving iTypes until a fixed point is reached (Line 9 - Line 25). The basic mechanism used by our approach is, for a node w in the iGraph, our approach computes all of the iTypes that can flow into it from its predecessors in the iGraph (Line 12). Then, our approach updates the iType information that will flow out of w by combining the incoming information with $Gen[w]$ and also updates the iGraph with any new edges that can be identified as a result of this. The specific manner of combining the iType information varies based on whether w is an sPoint (Line 15) or rPoint (Line 20). If w is a sending point (Line 15), our approach interprets each iType sent at w, using the iTypes defined in $In[w]$ via the *interpret* function to fill in the ICC placeholders (Line 17). Essentially this function acts as the transfer function and the substitution is done by matching Intent attribute names and references. The newly updated Intent information is then propagated to w's successors (Line 18). Then our approach creates new edges in G from w to all rPoints identified as target components (Line 19). If w is a receiving point (Line 20), our approach propagates the information flowing in w to its successors without making any changes (Line 21). This represents the semantics of an rPoint, which does not modify the Intents it receives. Our approach then adds edges from w to all sending points, s in the same component where an iType at s depends on information from w. This dependency can be identified in our approach by looking at the model representing the attributes of iTypes at s and determining if any have placeholders originating from Intents defined at w. The `ifIccDep` function represents this identification process, and checks all Intent type models at s to see if any contain a placeholder that originates from an Intent defined at a receiving point w.

Referring back to our running example, there are three sPoints in Program 1 (Lines 5, 12 and 17). Only the iType sent at Line 5 has no placeholders, and after its interpretation, results in an edge from the node representing Line 5 of Program 1 to the node representing Line 17 of Program 2 labeled with { ⟨`PACKAGE:{"com.example.app"}`⟩, ⟨`COMPONENT:{"com.example.app.A2"}`⟩ } . After this, iTypes sent at Lines 17 and 22 of Program 1 can be resolved. The placeholders in the iType sent at each line comes from Intent `res`. The iType for Line 17 is $\{\langle \epsilon_{id}, "2"\rangle, \langle \texttt{RES}, \text{requester component}\rangle\}$. The iType at Line 22 is modeled the same except "2" is "1". When Line 17 is under analysis, the In set would contain the received Intent information that is computed before. Therefore, the requester component would be resolved as $A1$. The `ifICC` function uses this information and adds an edge from the node representing the Line 17 to the node representing Line 7 and labels this edge with $\{\langle \epsilon_{id}, "2"\rangle, \}$. However, when Line 22 is under analysis, the In set would be empty. This results in no edge from this line getting added to the iGraph. { ⟨`ACTION:{"B_ACTION_SEND"}`⟩ ⟨$\epsilon_{\texttt{url}}$`:{getUrl}`⟩ ⟨$\epsilon_{\texttt{class}}$`:{"com.example.app.S"}`⟩ ⟨$\epsilon_{\texttt{id}}$`:{"2"}`⟩} An Intent with this content adds an edge from Line 12 of Program 1 to Line 3 of Program 2 to represent this ICC. Given the received Intent, rB, at Line 3, the Intent, rN, can be computed. First an edge from nodes representing Line 3 to the node Line 14 of Program 2 is added to the iGraph. Then, the `COMPONENT` of rN is computed as `"com.example.app.S2`, which results in starting $S2$. Eventually, the iGraph of our running example is computed.

Our approach terminates when the Out sets reach a fixed point, which means that all such dependencies have been resolved or no further updates are possible. A fixed point exists because the upper bound on all Out sets is the set of all possible iTypes that can be sent by apps in the bundle, and the Out sets grow monotonically since no iType information is invalidated during this analysis.

IV. Evaluation

To assess the accuracy, impact, and runtime of our approach, we conducted an empirical evaluation to address the following research questions:

RQ1: How accurate is our approach on Intent analysis benchmarks?
RQ2: How does component-sensitivity impact the analysis of real-world apps?
RQ3: What is the analysis time of our approach?

A. Implementation

For the purpose of the evaluation, we implemented a prototype, Accurate Component Sensitive Intent Analysis ($\mathcal{A}ria$), of our approach in Java. The intra-component and inter-component analyses were implemented using analysis primitives provided by the Soot analysis framework [30]. We implemented the `model` function in Algorithm 1 and `interpret` function in Algorithm 2 as extensions of the well known String analysis framework, Violist [29] to model the content of Intents as iTypes and interpret them. $\mathcal{A}ria$ will be made available to the community via our group's GitHub repository [31]. To the best of our knowledge, no other Intent analysis accounts for component-sensitivity. Therefore we chose to compare against a recent version of IC3 [32], and ICCBot [33] in order to baseline the performance of state of the art techniques when applied to Intents built from ICC information. We chose ICCBot as it was shown in a recent study to be the most accurate tool for computing ICCs within an app [25]. We chose IC3 as a comparison since it is widely used for Intent computation in the research community and serves as the underlying Intent computation mechanism for many other well-known ICC analysis approaches. These other approaches, specifically RAICC [34], Primo [35], StoryDroid [3] and DialDroid [36], extend IC3 to handle different types of Intent sources (e.g., fragments and atypical ICC links). These other sources do not account for component-sensitivity, and would therefore be complementary to our approach as opposed to overlapping. All experiments were conducted on an 8-core machine with an Intel i7 3.6 GHz CPU, 32GB of memory running Ubuntu 18.04 LTS 64-bit.

Info		Intent (Precision/Recall)			ICC Link (Precision/Recall)		
Sub	A/C	$\mathcal{A}ria$	IC3	ICCBot	$\mathcal{A}ria$	IC3	ICCBot
B1	2/6	1/1	0.75/0.75	1/0.5	1/1	0.43/1	1/0.67
B2	2/8	1/1	0.67/0.67	0/0	1/1	0.4/1	0/0
B3	2/8	1/1	0.67/0.67	1/0.33	1/1	0.5/1	1/0.33
B4	2/5	1/1	0/0	0/0	1/1	1/1	1/0.5
B5	3/8	1/1	0.5/0.33	1/0.33	1/1	0.33/1	1/0.33
B6	1/2	0.33/1	0/0	0/0	1/1	1/1	1/1
D1	3/3	1/1	0.5/0.2	0/0	1/1	0.57/1	0/0
D2	5/7	1/1	1/1	1/0.5	1/1	1/1	1/1
D3	3/3	0.67/0.5	1/0.5	1/0.5	1/0.5	1/0.5	1/0.5
D4	4/6	1/1	1/1	1/0.5	1/1	1/1	1/1
Avg	2.7/5.6	0.9/0.95	0.6/0.51	0.6/0.27	1/0.95	0.72/0.95	0.8/0.53

TABLE I: Benchmark subjects and RQ1 results

B. RQ1

The goal of this RQ is to assess the accuracy of our approach in performing Intent-based ICC analysis. To determine its accuracy, we ran $\mathcal{A}ria$ on a set of benchmark apps and measured its precision and recall. Our benchmarks came from DroidBench v3.0 [37], specifically the subset associated with the InterAppCommunication and InterComponentCommunication categories. Altogether, this subset comprised 15 apps (containing a total of 19 components) and we used these to create four app bundles. These are denoted in Table I with a "D" prefix. We considered including apps from ICCBench [38], but found that the benchmarks in DroidBench v3.0 were very similar or subsumed the ones in ICCBench. We created additional benchmark apps that included Intents created from ICC information since examples of this practice were not in either DroidBench or ICCBench. Altogether we created 12 new benchmark apps (containing a total of 37 components) and we used these to create six new app bundles. These are denoted in Table I with a "B" prefix. For each bundle, we manually analyzed the apps to identify the ground truth of ICC and then confirmed this by systematically executing the apps in the context of their bundle. Information about the benchmark bundles from both sources is shown in Table I. The first column lists the bundle identifier and the second column shows the number of apps (A) and the total number of components (C) in each bundle.

We ran $\mathcal{A}ria$, IC3, and ICCBot on each of the benchmark bundles and compared their output against the ground truth. We computed precision and recall for both Intents and ICC links. For Intent accuracy, we considered an Intent correctly identified if the tool's output matched the corresponding ground truth Intent for all attribute names and values. (Note that an over-approximation of an attribute value was considered an incorrect match.) For ICC link accuracy, we considered a link to be correctly identified if the tool's output matched the sPoint to the correct target component. The results for RQ1 are shown in Table I. For each benchmark bundle, the two columns labeled "Intents" and "ICC Links" show precision and recall (in the form precision/recall) for each of the three approaches. As seen in Table I, our approach more accurately calculated both Intents and ICC links in the benchmark bundles. This table displays the precision and recall calculated for each subject with respect to Intent computation and ICC link results. To determine the average accuracy of each approach, we calculated their respective F-measures. This involved first computing the average recall and precision for Intent and ICC link computations separately for each approach and then using these values to calculate the F-measure. In terms of Intent accuracy, our approach reported an F-measure of 92%. While this number for IC3 and ICCBot was 55% and 37%, respectively. Similarly, our approach achieved an F-measure of 97% for ICC links, whereas IC3 and ICCBot obtained F-measures of 82% and 64%, respectively. Taken together, these scores show the significant accuracy improvements that can be realized by our approach.

Both IC3 and ICCBot take very different approaches towards handling unknown values, and the implications of this were evident in the RQ1 results. Since neither approach attempts to handle Intent values defined using ICC information, this resulted in attribute names and values that could not be computed by these analyses. Typically, IC3 would over approximate unknown values to ".*" This resulted in it generally having higher recall than precision. Conversely, ICCBot typically does not include Intents in its output that have unknown values, leading to higher precision but lower recall. We further investigated ICCBot to understand if this observed lower recall could be due to its design for an intended usage scenario of reporting ICCs within a single app. To evaluate this, we randomly selected three bundles (B1, B2, B5), and for each, we manually created an alternate version that combined all apps in the bundle into a single app. Similar to the original results, ICCBot ignored the Intents constructed from ICC information in these bundles, too. However, with this change ICCBot's recall for ICC links improved to 58% from 53% on the original bundles, giving it a new F-score of 68%. Both of these scores were still significantly lower than our approach, so this result did not change our conclusion.

C. RQ2

The goal of RQ2 is to evaluate the impact of component-sensitivity on the ICC analysis of real-world apps. As subjects, we created 10 bundles each consisting of 40 apps. We chose a bundle size of 40 since that is similar to the average number of apps installed on a typical person's phone [39]. To create bundles, we collected 390 unique real-world apps and divided them into ten bundles, each containing 39 apps and one artificially generated app for simulating Android-specific

Intents. The artificial app simulated the Intents sent by the Android OS and included the ICCs resulting from the OS in our analysis (as outlined in Section Section III-A). For each bundle, we systematically implemented the app by defining an activity that sends out: (1) the standard protected Intents provided by Android, which were the same for all bundles, and (2) launching Intents for each app, which were extracted from the app's manifest file. The 390 real-world apps were randomly selected from AndroZoo [40]. We ensured that all apps can be analyzed by all tools. The selected subject apps varied in size: 31% of them were smaller than 1MB, 56% of them were between 1MB and 10MB, and 12% of them were larger than 10MB. To carry out this experiment, we ran $\mathcal{A}ria$ on subjects and measured its impact on identifying Intent-related information.

For RQ2, we were not able to compare directly against IC3 and ICCBot. For ICCBot, the reason for this was that it timed out for at least two apps in each of our bundles (a 30 minute limit was used, which is the same limit used by in its evaluation [22]), making its results incomplete. Also, by design, ICCBot reports intra-app ICCs, which again impacts the completeness of its results. For IC3, the obstacle was that we could not isolate the impact of component-sensitivity when comparing it against $\mathcal{A}ria$. The reason for this was that IC3's string analysis and call graph construction would both impact the accuracy of its results, but could not be replaced without extensive refactoring of its codebase. Therefore, they would be confounding factors for the validity of our conclusions. To address this, we instead approximated the performance of IC3 by running a version of our analysis without component-sensitivity (i.e., the inter-component analysis *does not* use ICC information in interpreting iTypes.) The validity of this approximation assumes IC3 would achieve similar accuracy as our intra-component analysis, a reasonable assumption given our observations in RQ1, and equalizes the impact of string analysis.

To measure impact, we ran $\mathcal{A}ria$ on each bundle and identified each Intent that contained an attribute name or value defined via ICC information. We did this by examining the intermediate artifacts produced by our analysis (i.e., the models representing Intent contents). Then, we examined these Intents to check if the attributes defined using ICC information were resolved to concrete values or if they had to be over-approximated. We also confirmed the accuracy of these results, using the same strict matching standard used in RQ1, by manually examining a random sample of Intents with a size large enough to give us a 95% confidence level with a 5% margin of error. This manual analysis involved analyzing control and data flow paths in methods, following call chains in each app, and also tracking ICCs and the data they carried in their bundles. Note that we only calculated precision since we were not confident in the accuracy of a manual analysis of such a large code base to identify all possible Intents and links.

Overall, we found a total of 601 Intents that use ICC information in their content (2% of the total number of computed Intents). These Intents had a total of 4,506 attributes, of which 2,357 were defined, at least in part, by a value from another Intent. $\mathcal{A}ria$ was able to compute concrete values for 90% (2,117/2,357) of the attributes. Our manual analysis calculated a precision rate of 94% in the random sample of Intents. The impact of accounting for ICD can be seen in these results. Within the 601 Intents, 52% (2,357/4,506) of their attributes had Inter Component Dependency, meaning that a significant portion of the attributes would be unknown without our approach's component-sensitivity. Moreover, the specific attributes were generally of high importance for understanding the semantics of ICCs. In nearly half of the Intents, the attribute setting the destination of the Intents that carry the result of a component's request become known. Also, in 214 Intents, the entire content of Intents would be unknown without resolving the relevant ICC information. Similarly, in 68 Intents the resolution of the ICDs of `COMPONENT` determined if an Intent was implicit or explicit for the purposes of ICC link matching. Also, we found that in total 739 `DATA` and `EXTRA` attributes, which are mainly used for passing information in apps for different functionalities (e.g., part of UI), had ICD, and $\mathcal{A}ria$ was able to find concrete values for 86% of them. The resolution of `EXTRA` attributes is particularly important for security analyses, since these attributes can be used to carry custom data between apps. Altogether, these results show that component-sensitivity is important for resolving a large number of Intents and obtaining important information about their contents.

	Benchmark Bundles (seconds)			Real World Bundles (minutes)		
	IC3	ARIA	ICCBot	IC3*	ARIA	ICCBot*
Min	36	18	6	24	10	13
Avg	109	36	17	36	16	24
Max	190	59	30	53	30	50

* excluding the apps that had a timeout at thirty minutes

TABLE II: Runtime of the three approaches analyzing bundles of apps (RQ3)

The impact of taking component-sensitivity into account can also be seen in comparing $\mathcal{A}ria$'s results indirectly with ICCBot and IC3. By definition, and also confirmed via the results in RQ1, we know that ICCBot does not include Intents with unknown values in its output and values defined via ICC would present as unknown. Therefore, for ICCBot, none of these 601 Intents would have been included in its output. This would also mean that 1,246 links identified by $\mathcal{A}ria$ would not have been identified. For IC3, these 601 Intents would have been present in its output. However, without component-sensitivity, all 2,117 values defined using ICD that were resolvable to concrete values by $\mathcal{A}ria$ would have been over-approximated as ".*". This over-approximation would have led to the identification of 4,087 spurious ICC links. Both ICCBot's under-approximation and IC3's over-approximation can be impactful as they could introduce inaccuracies into higher-level analyses that use their ICC information.

D. RQ3 - Analysis Time

To answer this RQ, we measured the time needed for running the analyses done as part of RQ1 and RQ2. To provide a point of comparison for *Aria* with respect to real world apps, we ran IC3 and ICCBot on RQ2's subjects, as well. For all tools, we set a 30-minute runtime limit for analyzing an individual app in the bundle but did not set any limit for the total runtime of a bundle. Results are shown in Table II. For the RQ1 benchmarks, the average analysis time per bundle was 36 seconds for *Aria*, 109 seconds for IC3, and 17 seconds for ICCBot. For the RQ2 real-world subjects, the average runtime per bundle of *Aria* was 16 minutes (ranged from 10 mins to 30 mins), IC3 was 36 minutes (ranged from 24 mins to 53 mins), and ICCBot was 24 minutes (ranged from 13 mins to 50 mins). *Aria* successfully analyzed all real-world apps, while IC3 timed out for one app, and ICCBot timed out for 38 apps. We excluded the timed out apps in the above-reported timing results, but if we did include their 30 minute timeout, the average runtime would be 39 minutes for IC3 and 138 minutes for ICCBot. When breaking down the total execution time of *Aria*, the intra-component phase took, on average, 13.5 minutes to analyze all 40 apps of the bundle, and the inter-component phase took 2.5 minutes. On average, computing models for Intents took 51% of the intra-component phase. While all tools were relatively fast in analyzing benchmark apps, *Aria* was both the fastest and most broadly applicable in analyzing real-world bundles.

Overall, these results show that *Aria* can successfully and efficiently analyze bundles. We believe that the efficiency of our approach can be attributed to (1) representing Intent attributes as String-based models, as opposed to expensive-to-solve systems of constraints and (2) the deferral of concretization of unknown Intent attributes through the use of placeholders in String models until all relevant Intent information is propagated, which reduces the amount of memory needed at runtime and avoids timeouts.

E. Threats to Validity

A potential threat is that a majority of the benchmark subjects were developed by us. This was necessary since the currently available benchmarks lacked instances of Intents constructed from ICC information. To mitigate this threat, we included four app bundles from DroidBench and designed our benchmarks based on real-world cases that we observed in apps. Another threat is that the real world app bundles may not be representative. To make the selected apps representative, we selected a large pool of apps, downloaded from different repositories, and we selected apps with different sizes, from 3.6 MB to 49 MB. Also, we ensured that all these apps contained common Intent usages. A threat to the validity of our study is that we used Violist to perform string analysis. Therefore, our work would suffer from Violist's limitations as related to loop unraveling. However, there exist no other available string analysis tools that can accurately and efficiently handle a wide range of string operations while providing us with the flexibility of defining new semantics. A potential threat to the validity of our results is that the effectiveness of our approach in computing Intents depends on the list of the Intent-related APIs that we collected. To mitigate this threat, we examined 400 real-world apps to collect the relevant API. Also, our approach is extensible, and new semantics can be added by defining the appropriate models. Another threat is that we could not compare the results of our approach on real-world apps against IC3 and ICCBot (i.e., RQ2) due to implementation limitations of those tools. However, to mitigate this challenge, we ran our approach on the same apps with different configurations to simulate these tools' behaviors.

V. Limitations

In this section, we discuss several limitations of our approach. (1) Our approach currently only handles Intent objects, but functions `model` and `interpret` can be extended to represent and interpret other ICC objects (e.g., PendingIntent) using the same AST-like model. (2) Our approach does not handle Extra pairs that are not defined using Strings. Although our experience is that this is the most common way of defining these attributes, our approach could be extended to support other types by modeling them as strings. (3) The string analysis capability of our approach is limited to Violist's capability. Although our experiments show that this imposes few limitations in practice, one limitation is modeling attribute names or values that are defined in loops, since Violist models the semantics of loops by unrolling them a fixed number of times. (4) Our approach does not handle native code and it over-approximates values that are provided during runtime, such as network values. In general, we see these limitations as addressable by our approach, as defined in Section III, but that would require additional software development effort.

VI. Related Work

A large body of research focuses on Android ICC analysis, both at the intra-app and inter-app levels using static analysis and dynamic analysis. Works such as StoryDroid [3], ICCBot [22], IC3 [19], EPICC [41], Gator [2] and RAICC [34] aim at extracting ICC information statically at the sending side. There are also some works, such as Fax [6] and Letterbomb [42], that focus on the receiving side. These works compute Intent information based on a set of constraints defined on Intents' content at the receiving side. However, these techniques do not account for the information that is received by components. Our approach computes ICCs with respect to the ICC information that can get propagated in a bundle of apps by connecting the sending and receiving sides to each other. RAICC models atypical ways of performing ICC, ICCBot, Gator, and StoryDroid model the UI relationships (e.g., fragments to Activities). While the current design of our approach does not handle these aspects of ICC analysis, our approach is extensible and can model new semantics. Also, the contributions of techniques such as RAICC [34] and Gator [2] can be complementary to our technique to get a larger and more complete picture of ICCs in a bundle of apps. Works such as Primo [35] and DialDroid [36] extended IC3 to become

more accurate by using statistical models and incremental callback analysis. However, they did not change the underlying analysis of Intent computation, so these approaches would also be complementary to our work.

Several techniques have ICC analysis as a part of their technique. These works either leverage existing tools, mainly IC3, (e.g., [7], [18], [43], [23], [44], [45]) or they model ICCs themselves (e.g., [5], [46], [47], [48]. The first group of techniques has limitations with respect to ICD due to their underlying ICC tool inaccuracies. The second group has a wide range of limitations. Results have shown that the ICC analysis used in existing techniques [49], [50] is not accurate for real-world app analysis. There are many reasons for inaccuracies in computing Intent-related ICCs [25]; however, in this paper, we introduced a technique that takes the inter-component related information into account when modeling Intents to compute more accurate ICC information.

Numerous program analysis techniques have been proposed to focus on security issues in the ICC model. Security techniques such as [51], [52], [53], [54], [14], [55], [56] use static analysis to perform ICC analysis within the boundary of one app. Techniques such as [10], [9], [11], [8], [57], [58], [13], [15], [59], [60], [12], [61] have been introduced to analyze ICCs across multiple apps. In addition to static analysis, a group of techniques leverage dynamic analysis to analyze ICCs [62], [63], [24], [64], specially for testing the robustness of apps. However, these techniques do not provide a complete picture of possible ICCs due to the inherent limitation of dynamic analysis.

VII. Conclusion and Future Work

The Android ICC model allows components within an app and across apps to communicate with each other to achieve goals such as screen transition or sharing information. Due to the widespread usage of this model, accurate information about ICC related objects is necessary for many static analysis techniques. To address the shortcomings of existing techniques, we introduced a multi-level component-sensitive static analysis technique to efficiently and accurately compute Intent information in bundles of apps. In the first phase, our approach models Intents that each component can send out with placeholders for information coming from ICCs, and then, in the second phase, it propagates ICC information among the components in the bundle to compute the content of Intents and ICCs. In comparison with state of the art, our approach was able to compute more accurate Intent-based ICC information in a shorter time. As future work, we plan to adapt our technique to compute non-Intent based ICCs and attributes of more complex types such as Arrays in the `EXTRA` attribute.

Acknowledgments

This work was supported, in part, by grant 2211454 from the U.S. National Science Foundation.

References

[1] Android, "Intent," Dec 2019, [Online; accessed 16. Aug. 2020]. [Online]. Available: https://developer.android.com/reference/android/content/Intent

[2] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for Android," in *IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 658–668.

[3] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 596–607. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00070

[4] S. Chen, L. Fan, C. Chen, and Y. Liu, "Automatically distilling storyboard with rich features for android apps," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 667–683, 2023.

[5] Z. Chen, J. Liu, Y. Hu, L. Wu, Y. Zhou, Y. He, X. Liao, K. Wang, J. Li, and Z. Qin, "Deuedroid: Detecting underground economy apps based on utg similarity," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 223–235. [Online]. Available: https://doi.org/10.1145/3597926.3598051

[6] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of android applications by constructing activity launching contexts," in *Proceedings of the 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, pp. 457–468.

[7] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 115–127.

[8] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

[9] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: detecting inter-component privacy leaks in Android apps," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Florence, Italy: IEEE Press, May 2015, pp. 280–291.

[10] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek, "Information flows as a permission mechanism," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 515–526. [Online]. Available: https://doi.org/10.1145/2642937.2643018

[11] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," *ACM Transactions on Privacy and Security*, vol. 21, no. 3, pp. 14:1–14:32, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3183575

[12] H. Bagheri, C. Tang, and K. Sullivan, "Automated synthesis and dynamic analysis of tradeoff spaces for object-relational mapping," *IEEE Trans. Softw. Eng.*, vol. 43, no. 2, p. 145–163, Feb. 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2587646

[13] Y. K. Lee, P. Yoodee, A. Shahbazian, D. Nam, and N. Medvidovic, "SEALANT: a detection and visualization tool for inter-app security vulnerabilities in Android," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, Oct. 2017, pp. 883–888.

[14] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 239–252. [Online]. Available: https://doi.org/10.1145/1999995.2000018

[15] A. Sadeghi, H. Bagheri, and S. Malek, "Analysis of android inter-app security vulnerabilities using covert," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 725–728.

[16] B. Wang, C. Yang, and J. Ma, "Iafdroid: Demystifying collusion attacks in android ecosystem via precise inter-app analysis," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 2883–2898, 2023.

[17] X. Deng, J. Yan, S. Zhang, J. Yan, and J. Zhang, "Variable-strength combinatorial testing of exported activities based on misexposure prediction," *Journal of Systems and Software*, vol. 204, p. 111773, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223001681

[18] Y. K. Lee, J. y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A sealant for inter-app security holes in android," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 312–323. [Online]. Available: https://doi.org/10.1109/ICSE.2017.36

[19] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 77–88.

[20] D. Octeau, D. Luchaup, S. Jha, and P. McDaniel, "Composite constant propagation and its application to android program analysis," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 999–1014, 2016.

[21] J. Samhi, A. Bartel, T. F. Bissyandé, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," 2021.

[22] J. Yan, S. Zhang, Y. Liu, J. Yan, and J. Zhang, "Iccbot: Fragment-aware and context-sensitive icc resolution for Android applications," in *The 44th International Conference on Software Engineering, ICSE 2022 (Demo Track)*, 2022.

[23] F. Pauck and H. Wehrheim, "Together strong: Cooperative android app analysis," ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 374–384. [Online]. Available: https://doi.org/10.1145/3338906.3338915

[24] H. Cai and B. G. Ryder, "Understanding android application programming and security: A dynamic study," *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 364–375, 2017.

[25] J. Yan, S. Zhang, Y. Liu, X. Deng, J. Yan, and J. Zhang, "A comprehensive evaluation of android icc resolution techniques," in *The 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, year = 2022,*.

[26] Android, "Application Fundamentals | Android Developers," Dec 2019, [Online; accessed 16. Aug. 2020]. [Online]. Available: https://developer.android.com/guide/components/fundamentals

[27] "Intent Resolution Rules," Jun. 2023, [Online; accessed 29. Jul. 2023]. [Online]. Available: https://developer.android.com/reference/android/content/Intent#intent-resolution

[28] Android, "IntentFilter | Android Developers," Dec 2019, [Online; accessed 16. Aug. 2020]. [Online]. Available: https://developer.android.com/reference/android/content/IntentFilter

[29] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, "String analysis for java and android applications," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, September 2015.

[30] R. Vall, e Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay, "Soot - a java bytecode optimization framework," 1999. [Online]. Available: citeseer.ist.psu.edu/vallee-rai99soot.html

[31] "Aria," 2023. [Online]. Available: https://github.com/USC-SQL/ARIA

[32] JordanSamhi, "Github - jordansamhi/ic3: Ic3: Inter-component communication analysis in android," 2022. [Online]. Available: https://github.com/JordanSamhi/ic3

[33] hanada31, "ICCBot," Nov. 2022, [Online; accessed 11. Nov. 2022]. [Online]. Available: https://github.com/hanada31/ICCBot

[34] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein, *RAICC: Revealing Atypical Inter-Component Communication in Android Apps*. IEEE Press, 2021, p. 1398–1409. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00126

[35] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable market-scale android inter-component analysis," *SIGPLAN Not.*, vol. 51, no. 1, p. 469–484, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2914770.2837661

[36] dialdroid android, "DIALDroid," Apr 2018, [Online; accessed 7. May 2020]. [Online]. Available: https://github.com/dialdroid-android/DIALDroid

[37] secure-software engineering, "DroidBench," Sep 2016, [Online; accessed 15. March 2021]. [Online]. Available: https://github.com/secure-software-engineering/DroidBench

[38] fgwei, "ICC-Bench," Mar 2017, [Online; accessed 15. March 2021]. [Online]. Available: https://github.com/fgwei/ICC-Bench

[39] A. Eira, "Number of Mobile App Downloads in 2022/2023: Statistics, Current Trends, and Predictions," *Financesonline*, May 2023. [Online]. Available: https://financesonline.com/number-of-mobile-app-downloads

[40] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[41] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. USA: USENIX Association, 2013, p. 543–558.

[42] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek, "Automatic generation of inter-component communication exploits for android applications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 661–671. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106286

[43] C. C.-C. Cheng, C. Shi, N. Z. Gong, and Y. Guan, "Logextractor: Extracting digital evidence from android log messages via string and taint analysis," *Forensic Science International: Digital Investigation*, vol. 37, p. 301193, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281721001013

[44] L. Li, A. Bartel, T. Bissyandé, J. Klein, Y. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings - 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, ICSE 2015*, ser. Proceedings - International Conference on Software Engineering. United States: IEEE Computer Society, 8 2015, pp. 280–291.

[45] Y. Wu, J. Shi, P. Wang, D. Zeng, and C. Sun, "Deepcatra: Learning flow- and graph-based behaviours for android malware detection," *IET Information Security*, vol. 17, no. 1, pp. 118–130, 2023.

[46] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, pp. 14:1–14:32, Apr. 2018. [Online]. Available: http://doi.acm.org/10.1145/3183575

[47] M. Gordon, K. deokhwan, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of android applications in droidsafe," 01 2015.

[48] J. Yan, X. Deng, P. Wang, T. Wu, J. Yan, and J. Zhang, "Characterizing and identifying misexposed activities in android applications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 691–701. [Online]. Available: https://doi.org/10.1145/3238147.3238164

[49] F. Pauck, E. Bodden, and H. Wehrheim, "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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 331–341. [Online]. Available: https://doi.org/10.1145/3236024.3236029

[50] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, Jul. 2018, pp. 176–186. [Online]. Available: https://doi.org/10.1145/3213846.3213873

[51] X. Cui, J. Wang, L. C. K. Hui, Z. Xie, T. Zeng, and S. M. Yiu, "WeChecker: efficient and precise detection of privilege escalation vulnerabilities in Android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '15. New York, New York: Association for Computing Machinery, Jun. 2015, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2766498.2766509

[52] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, p. 259–269, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[53] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: semantics-based detection of Android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. Hong Kong, China: Association for Computing Machinery, Nov. 2014, pp. 576–587. [Online]. Available: https://doi.org/10.1145/2635868.2635869

[54] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *NDSS*, 2012.

[55] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 229–240. [Online]. Available: https://doi.org/10.1145/2382196.2382223

[56] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek, "Automatic generation of inter-component communication exploits for android applications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 661–671. [Online]. Available: https://doi.org/10.1145/3106237.3106286

[57] "DidFail," Mar 2015, [Online; accessed 14. May 2020]. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=508078

[58] W. Klieber, L. Flynn, W. Snavely, and M. Zheng, "Practical Precise Taint-flow Static Analysis for Android App Sets," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018. Hamburg, Germany: Association for Computing Machinery, Aug. 2018, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3230833.3232825

[59] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.

[60] H. Bagheri, J. Wang, J. Aerts, and S. Malek, "Efficient, evolutionary security analysis of interacting android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 357–368.

[61] M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-An, and X. Luo, "Dina: Detecting hidden android inter-app communication in dynamic loaded code," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2782–2797, 2020.

[62] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 627–638. [Online]. Available: https://doi.org/10.1145/2046707.2046779

[63] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 118–128. [Online]. Available: https://doi.org/10.1145/2771783.2771800

[64] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "Intentfuzzer: Detecting capability leaks of android applications," ser. ASIA CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 531–536. [Online]. Available: https://doi.org/10.1145/2590296.2590316