ICCBot: Fragment-Aware and Context-Sensitive ICC Resolution for Android Applications

Jiwei Yan
Tech. Center of Software Engineering
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China
yanjw@ios.ac.cn

Shixin Zhang
School of Software Engineering
Beijing Jiaotong University
Beijing, China
lightn.rs@gmail.com

Yepang Liu
Dept. of Computer Science and Engr.
Southern University of Sci. and Tech.
Shenzhen, China
liuyp1@sustech.edu.cn

Jun Yan*
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China
yanjun@ios.ac.cn

Jian Zhang*
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China
zj@ios.ac.cn

ABSTRACT

For GUI programs, like Android apps, the program functionalities are encapsulated in a set of basic components, each of which represents an independent function module. When interacting with an app, users are actually operating a set of components. The transitions among components, which are supported by the Android Inter-Component Communication (ICC) mechanism, can reflect the skeleton of an app. To effectively resolve the source and destination of an ICC message, both the correct entry-point identification and the precise data value tracking of ICC fields are required. However, with the wide usage of Android fragment components, the entry-point analysis usually terminates at an inner fragment but not its host component. Also, the simply tracked ICC field values may become inaccurate when data is transferred among multiple methods. In this paper, we design a practical ICC resolution tool ***ICCBot***, which resolves the component transitions that are connected by fragments to help the entry-point identification. Besides, it performs context-sensitive inter-procedural analysis to precisely obtain the ICC-carried data values. Compared with the state-of-the-art tools, *ICCBot* achieves both a higher success rate and accuracy. *ICCBot* is open-sourced at **https://github.com/hanada31/ICCBot**. A video demonstration of it is at **https://www.youtube.com/watch?v=7zcoMBtGiLY**.

*Corresponding Authors.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Android App, Inter-Component Communication, ICC Resolution, Fragment, Component Transition Graph

ACM Reference Format:
Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. ICCBot: Fragment-Aware and Context-Sensitive ICC Resolution for Android Applications. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516864

1 INTRODUCTION

Android ICC is a framework-related mechanism, which plays an important role in building implicit control flow as well as transferring data among Android app components. The comprehensive extraction of ICC links can help to understand GUI structure and detect inter-component or inter-app risks. Many existing works [15, 19, 20] concentrate on the resolution of ICC links or the construction of Component Transition Graphs (CTGs). However, most of them only consider the scenarios that one basic component, i.e., the instance of Activity, Service, or Broadcast Receiver, directly launches another, but do not take the behaviors of fragment [8] components into consideration. According to existing research [17], nearly 91% of 217 top downloaded apps from Google Play use fragments. Also, some approaches adopt the context-insensitive strategy to improve efficiency, which leads to imprecise tracking of data transfer among method calls.

In this paper, we aim to resolve ICC links with consideration of both precision and efficiency. We implement a fragment-aware and context-sensitive ICC resolution tool, called *ICCBot*. For precision improvement, it first models the behaviors of fragments and extracts the fragment-loading graph, then connects the components linked through fragments. During analysis, it constructs summaries for each method according to the bottom-up topology order in the call graph, as well as analyzes the relationship between the callee's summary and values of calling context. When encountering a method call, only specific context values will be tracked to update the summary of the callee. If there is no summary-related context value, the method summary of the callee will be reused directly. According to the evaluation results, *ICCBot* achieves a 95% precision on two popular benchmarks and greatly outperforms the state-of-the-art tools on a benchmark containing fragment- and context-related ICCs. Besides, on large-scale real-world apps, it resolved 28,584 ICC edges and 11,871 fragment-loading edges on 2,000 real-world apps, in which the fragment and context-sensitive strategies bring over 8% and 25% ICC links.

2 BACKGROUND OF ICC RESOLUTION

In the Android system, the basic function of ICC mechanism is achieved by constructing and sending `Intent` objects in apps. An Intent is a messaging object users can use to request an action from another app component [10]. There are three fundamental use cases for Intent to interact with basic components, starting an activity, starting a service and delivering a broadcast. Each Intent object carries a group of data items that will be delivered to the Android system, parts of which are used by the system to determine the target component. Thus, the destination of an ICC transition can be obtained by precise ICC field extraction. Normally, developers send an ICC message in the lifecycle or user-callback methods of a basic component. Thus, the source of an ICC transition can be obtained by analyzing the entry-point method of Intent sending.

Nowadays, the `Fragment` [8] component, which is hosted by an activity or another fragment for reusable UI, is becoming more and more popular. It has its own lifecycle callback methods. Many developers choose to load fragments first and then send ICC messages in their callback methods. Therefore, to identify the actual source component of an ICC, the control flows between activities and the inner fragments should be built first, based on which we could extract the activity-level hosts of the fragments that actually trigger the Intent sending behavior.

Fig. 1 displays a motivating example with fragment loading and context-related ICC data assignments. In lines 7-9, the fragment `MyFragment` is created and loaded in activity component `MyActivity`. For the Intent object `i` in line 23, without fragment modeling, the entry-point tracking analysis will terminate at the fragment lifecycle method `onAttach()` but miss the actual entry in its host activity. In lines 15-18, when the fragment `MyFragment` is attached, the method `sendICC()` will be invoked with a specific action value. This context value is used to construct an `Intent` object, while different values passed to the callee will lead to different ICC targets. In line 24, the Intent object is passed into method `addCategory()` for category adding. Considering both the Intent object itself and values of data fields can be passed among methods, context-sensitive inter-procedural analysis should be performed. Finally, in line 25, the updated Intent is sent to Android system by API `startActivity()`. For this example, *ICCBot* can detect one fragment-loading edge and two component-transition edges, while the related tools *IC3* [19], *IC3DIALDroid* [16], and the ATGClient of *Gator* [5] report no ICC.

```
1  public class MyActivity extends Activity {...
2     protected void onCreate(Bundle savedInstanceState) {
3        Button btn = (Button)findViewById(R.id.button);
4        btn.setOnClickListener(new OnClickListener(){
5           //Callback Entry of Component
6           public void onClick(View v){
7              getSupportFragmentManager().beginTransaction()
8              .replace(R.id.fragment, new MyFragment())
9              .commit();    //Fragment Loading
10          }
11 });}}
12 public class MyFragment extends Fragment {...
13    public void onAttach(Activity activity) {
14       //Callback Entry of Fragment
15       if(...)
16          Utils.sendICC("action.first");
17       else
18          Utils.sendICC("action.second");
19 }}
20 public class Utils{
21    public static void sendICC(String mAction){
22       //Call Context Related ICC
23       Intent i = new Intent(mAction);
24       addCategory(i);
25       getActivity().startActivity(i);
26    }
27    public static void addCategory(Intent i){
28       i.addCategory("category");
29    }
30 }
```

Figure 1: Motivating Example

If the fragment is removed, these existing tools all generate non-existing ICCs when method `sendICC()` is called in multiple components, due to their context-insensitive analysis.

3 ICCBOT: FRAGMENT-AWARE AND CONTEXT-SENSITIVE ICC RESOLUTION

This section first displays the overview of *ICCBot* and then mainly introduces the extraction of fragment-aware transitions and the construction of context-sensitive summaries.

3.1 Overview of ICCBot

Fig. 2 displays the overview of our ICC resolution approach, which takes the apk file and configuration parameters as input and outputs the visualized graphs and other resolution-related results. It contains four basic modules: 1) entry-point-related control-flow analysis, 2) ICC-field-related data-flow analysis, 3) Intent/Fragment object modeling, and 4) summary construction module.

For the **control-flow analysis**, we first use *FlowDroid* [14] to get the initial call graph (CG) and callback entries. Due to the lack of asynchronous call relations, we maintain a mapping of commonly used asynchronous pairs and search the asynchronous callee on-the-fly according to the inferred signature of callees. And for

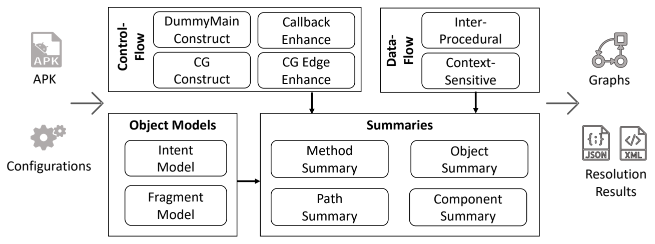


Figure 2: Framework of *ICCBot* (the CTGClient)

polymorphism-related invocations, we try to identify the correct callee by tracking the allocation sites of reference variables and extracting types of the newly created objects. Moreover, we extend the callback entrance set by identifying the user-customized callback listeners. For **data-flow analysis**, we adopt the intra-procedural reaching definition analysis to get the target variable related *du-chains* [13] in each method. Thus, we can track the assignments of each ICC field, like the `val` in `i.setAction(val)`, as well as the key declaration and value of extra data items, e.g., the `key` and `val` in `i.putExtra(key, val)`. For data values that come from calling contexts, e.g., the formal parameter or return value, we perform an inter-procedural context-sensitive analysis to track the data values passing among methods. For the **object modeling module**, both the Fragment and Intent objects are concerned. The fragment model describes the fragment creating, adding, and replacing behaviors, and the Intent model describes the Intent creating, packaging, and sending behaviors. We model the behaviors of the related APIs according to their semantics and identify the objects whose behavior successfully triggers fragment loading and Intent sending. During analysis, we construct **multiple-level summaries** for each method, intra-procedural path, and object instance to describe the ICC behaviors, and collect the ICC source and destination information to build the component summaries. Finally, *ICCBot* outputs component transition graph, fragment loading graph, or other information according to users' configuration.

3.2 Fragment-Aware Transition Extraction

There are two ways to start a fragment component in Android apps. For the static loading, the activity component can declare the fragment inside its layout file. Thus, by analyzing the layout declarations and the view inflating statements, we can get corresponding $\langle act, frag \rangle$ pairs. Besides, the activity component can add fragments into an existing `ViewGroup`. For example, we can obtain the `FragmentManager` and submit a transaction to it, which includes a set of fragment-related operations, like *add*, *replace*, etc. The actual fragment loading operations are executed by the Android framework, which is invisible in CG. To extract these edges, we model the fragment operating APIs and then infer the fragment behaviors by tracking the transaction sequences committed into the `FragmentManager`. A valid fragment transaction sequence starts from the creation of an `FragmentTransaction`, contains at least one fragment modification operation, and ends with the transaction commit operation. By further analyzing the modification targets, we can get either $\langle act, frag \rangle$ or $\langle frag, frag \rangle$ pairs. And the invalid operation sequences, e.g., not committed transactions, will be dropped. Finally, by analyzing the transitive relation among activities and fragments, we point out the activity-level host set of each fragment in pair $\langle frag, S_{act} \rangle$. Based on the analysis, when the callback entrance that belongs to a fragment is reached, *ICCBot* could quickly identify its actual activity-level host component.

3.3 Context-Sensitive Summary Analysis

According to literature [16], the reliability and the usability of tools are important concerns when researchers analyze complex apps. To improve efficiency, we consider building a set of lightweight context-sensitive summaries that reduce unnecessary computation.

Method Summary. To perform inter-procedural analysis, we first build acyclic CGs in which entry methods are the root nodes. For each callback entrance *m*, we traverse the edges *e* in the original CG$_{original}$ whose source is *m*. If there is no path from *e.target* to *m* in the new CG$_{acyclic}$, then add *e* to CG$_{acyclic}$ and traverse the *e.target* recursively. The following analysis is in a bottom-up topology order according to the acyclic CG, which means the callee is always analyzed earlier than its caller. The summary of a method can be formally defined as a triple $\mathcal{MS} = \langle mtd, S_{os}, S_{ms} \rangle$, where

- mtd is the unique signature of current method.
- S_{os} is a set of object summaries. Each $os \in S_{os}$ is an object summary whose object $os.obj$ is created or received in mtd.
- S_{ms} is a set of method summaries. Each $ms \in S_{ms}$ denotes a callee method that is invoked by mtd. The inner object summaries in $ms.S_{os}$ are all context-irrelevant with their caller, i.e., they will not be influenced by any context information in mtd.

Each method summary contains a set of object summaries and a set of callee method summaries. The object summary set S_{os} describes the created and received Intent or Fragment objects in the current method. For the callees whose object summaries have relationships with the invocation context, we efficiently build new object summaries into S_{os} according to callee's method summary and their relationships. And for other callees whose object summaries are context-irrelevant with method mtd, their method summaries will be directly added into S_{ms} to avoid re-analysis. For each method, the summaries are path-sensitively built on the Intent or Fragment-related code slices, and the maximum number of paths under analysis is limited by a threshold to avoid path exploration.

Object Summary. The object summary models the behaviors of an Intent or Fragment object by analyzing the operating statements of an object in one path. Both the Intent and Fragment-related statements are grouped into three categories, i.e., *create*, *update* and *send*. For a valid object summary, each category must contain at least one statement. We manipulate the object instance according to the semantic of each statement. For example, the statement `i = new Intent()` creates Intent object `i` and the statement `i.addAction(s)` updates the action field of `i`. The valid summary of an Intent or Fragment object can be formally defined as a 4-tuple $\mathcal{OS} = \langle obj, mtd, L_{stmt}, M_{ctx} \rangle$, where

- *obj* is an object instance to be modeled, which contains a set of fields according to its concrete type.
- *mtd* is the signature of method where *obj* is created or received.
- L_{stmt} is a list of Intent/Fragment related statements, where $L_{stmt} = L_c \cup L_u \cup L_s$. L_c is a list of object creating or receiving statements that creates object *obj*, $|L_c| = 1$; L_u is a list of object updating statements that updates the fields of object *obj*, $|L_u| \geq 1$; L_s is a list of object sending statements, which delivers object *obj* to Android framework, $|L_s| = 1$. When a new statement *s* is observed and added into L_{stmt}, object *obj* will be modified according to the extracted target field and the filed value in *s*.
- M_{ctx} records the map from object fields to context locations, e.g., for the Intent summary created in method `sendICC()`, pair $\langle action, sendICC_1 \rangle$ means the value of the action field is influenced by the first formal parameter of `sendICC()`. This pair will be transitively updated when its caller is analyzed and the first actual parameter of `sendICC()` is also context-related.

Component Summary. Based on object summaries, we can obtain the Fragment loading edges to enhance CG and extract the source/destination information to resolve ICC links. Then we analyze all the Fragment and Intent object summaries to further build component summaries, which can be formally defined as a tuple $CS = \langle src, S_{des} \rangle$, where

- src is the source component that sends out the ICC message.
- S_{des} is a set of destination components to be loaded or launched.

4 USAGE

Multiple Clients. *ICCBot* provides a group of clients as follows. Users can customize their own clients based on these default clients or just use them directly.

- **CallGraphClient**. Output the enhanced CG of an app and the topology order of methods, including the asynchronous and polymorphic methods related edges.
- **ManifestClient**. Extract the AndroidManifest.xml file in app and parse the manifest information.
- **IROutputClient**. Invoke Soot [12] to write the Jimple IR files.
- **FragmentClient**. Extract the fragment-loading relationships, including visualized graphs and the detailed summary files.
- **CTGClient**. The default client. Analyze the component transition relationships, including visualized graphs and the summary files of each Intent-sending and fragment-loading edge.
- **ICCSpecClient**. Output the inferred ICC specifications for components by extracting the component declaration information and analyzing the send and receive-related ICC messages.

Customized Configuration. *ICCBot* provides various configuration items to customize the analysis process. For the default *CTGClient*, the normal configuration items such as *maxPathNumber* (for intra-procedural path-sensitive analysis) and *maxAnalyzeTime* could be adjusted for the scalability. Meanwhile, the analysis can be performed while including or excluding specific Android characteristics, like fragment loading, context-sensitive analyzing, CG enhancing, String API analysis, etc.

Friendly Output. The outputs for both the *CTGClient* and *FragmentClient* are visualized with dot graphs. Fig. 3 and Fig. 4 give parts of the CTG and Fragment loading outputs of app *CSipSimple* [1], in which both the component transitions and the fragments-loading edges are displayed. And Fig. 5 gives a snippet of report generated by the *ICCSpecClient*, which is in JSON format and displayed with the help of a JSON viewer [11]. For each component, we give the action, category, data, type and extra data specifications, in which the intent-filter information declared in manifest, the received ICC-field information from another component, and the ICC values sent out from current component are all given.

5 EVALUATION

By searching the tools that are publicly available for ICC resolution, we select three state-of-the-art tools *IC3* [2], *IC3-DIALDroid* [6] and *Gator (ATGClient)* [5], and then compare their ICC resolution ability with *ICCBot* [9]. We evaluate these tools on two types of benchmarks. One is formed by a set of hand-made apps, and the other contains large-scale real-world apps.

First, we collect two well-known benchmarks, the *DroidBench* [3] (ICC-related part) and the *ICC-Bench* [4], which contain 38 ICC links in total. In addition, we design a compact self-made benchmark *ICCBotBench* [9], which contains two fragment-related ICCs that cover the basic static and dynamic fragment loading usages, seven context-related ICCs that try to pass the target objects or related values among methods, and two ICCs that involve both the fragment and context characteristics.

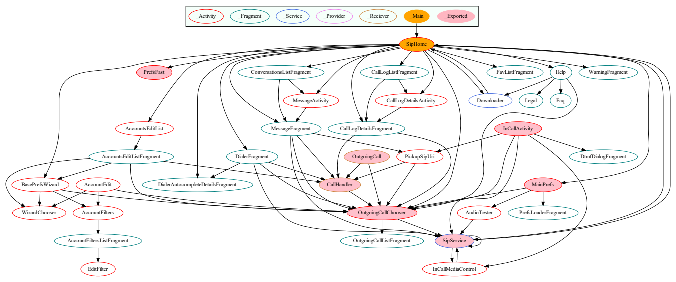


Figure 3: The Output of CTGClient

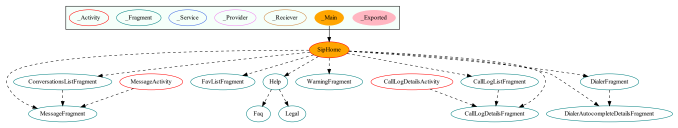
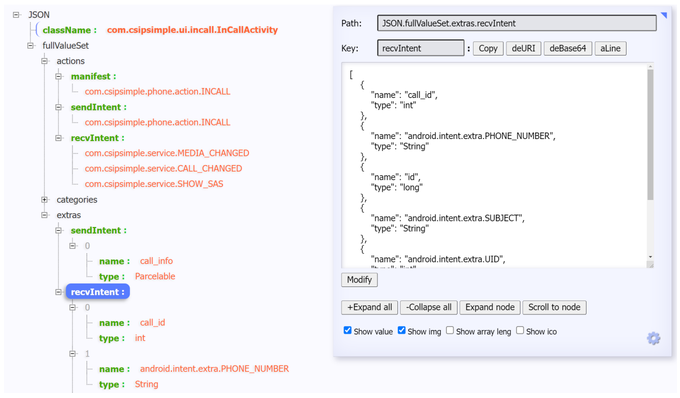


Figure 4: The Output of FragmentClient


```
JSON
  className : com.csipsimple.ui.incall.InCallActivity
  fullValueSet
    actions
      manifest :
        com.csipsimple.phone.action.INCALL
      sendIntent :
        com.csipsimple.phone.action.INCALL
      recvIntent :
        com.csipsimple.service.MEDIA_CHANGED
        com.csipsimple.service.CALL_CHANGED
        com.csipsimple.service.SHOW_SAS
    categories
    extras
      sendIntent :
        0
          name :  call_info
          type :  Parcelable
      recvIntent :
        0
          name :  call_id
          type :  int
        1
          name :  android.intent.extra.PHONE_NUMBER
          type :  String
```

Path: JSON.fullValueSet.extras.recvIntent

Key: recvIntent

```
[
  {
    "name": "call_id",
    "type": "int"
  },
  {
    "name": "android.intent.extra.PHONE_NUMBER",
    "type": "String"
  },
  {
    "name": "id",
    "type": "long"
  },
  {
    "name": "android.intent.extra.SUBJECT",
    "type": "String"
  },
  {
    "name": "android.intent.extra.UID",
```

Figure 5: The Output of ICCSpecClient

The corresponding evaluation results are given in Table 1. It presents the behaviors of tools on different benchmarks, in which the second column is the number of ground truth ICC links in each benchmark. The following columns are the number of true positive (#TP) and false positive (#FP) ICCs of each tool. The number of false negative (FN) ICCs can be computed by $\#GT - \#TP$. According to the results, the off-the-shelf tools suffer from both the FN and FP, where *ICCBot* can correctly resolve ICC in almost all cases. By further investigation, we find that the dismissing of characteristics like fragment and incomplete callback handling are the key reasons leading to FN. The context-insensitive analysis is the

Table 1: Evaluation on Hand-made Benchmarks

Benchmark	#GT	#TP (#FP)			
		IC3	*IC3*$_{dial}$	*Gator*	*ICCBot*
DroidBench	12	8 (0)	8 (0)	7 (1)	10 (0)
ICC-Bench	26	10 (0)	23 (0)	4 (0)	26 (0)
ICCBotBench	11	7 (3)	10 (24)	7 (3)	11 (0)
Sum	49	25 (3)	41 (24)	18 (4)	47 (0)

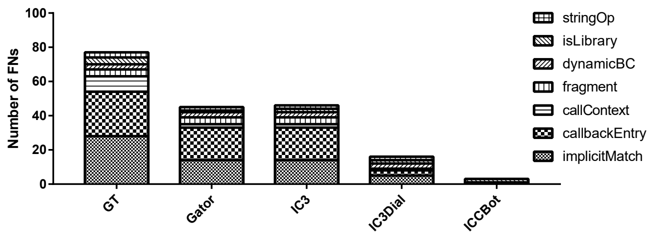


Figure 6: Number of FNs related to each Characteristic

main reason for FPs in three tools, especially for *IC3-DIALDroid* that context-related FPs are caused by the side effects of entry analysis optimization. Though no FP is reported by *ICCBot*, FPs are still possible for *ICCBot* takes the top method that it could track as the entry-point when the complex callback registrations are finally missed, which may bring errors. Then we summarize the ICC-related key characteristics used in the three benchmarks, including 4 fragments, 9 call contexts, 26 callback entrances, and 38 other characteristics related ICCs, and count the number of ICCs related to them. In Fig. 6, the first bar gives the number of ground truth (GT) ICCs about each characteristic. And the others give the FN ICCs related to each summarized characteristic, which reflects the differences of tools on various code characteristics.

To further validate the effectiveness of tools on real-world apps, we perform evaluations on 2,000 apps from F-droid [7]. For all the tools, we set 30 minutes as the analysis time threshold for each app. Table 2 shows the number of apps that succeeded and failed (crash or timeout) to be analyzed, the average analysis time for all the successfully analyzed apps, the number of the detected ICC links, and the number of fragment-loading edges. According to the results, *ICCBot* achieves a higher success rate than other tools with acceptable efficiency. Considering the existence of FP and FN in the state-of-the-art tools, the number of ICC links is only provided for reference. As *ICCBot* is configurable, we compare its default configuration with the ones that exclude fragment and context-sensitive data analysis. Our context-insensitive strategy omits to track the data transfer among methods instead of merging data from all the contexts, so that will not bring FPs. According to the results, the fragment analysis added 11,871 edges in the CTG for fragment displaying. And the two strategies contribute 2,284 (8.68%), and 5,769 (25.29%) ICCs, respectively, which greatly improves the completeness of the original CTG.

Table 2: Evaluation on Real-world Benchmarks

Tool	#Success	#Fail	Time$_{succ}$	#ICC	#Fragment
IC3	1,719	281	57s	10,860	-
IC3$_{dial}$	1,851	149	115s	8,601	-
Gator	1,882	118	20s	27,297	-
ICCBot	2,000	0	54s	28,584	40,455

6 RELATED WORK

For the task of ICC resolution, *Epicc* [20] and *IC3* [19] model the Intent-related APIs and resolve the ICC field values by data-flow analysis. Nowadays, many works choose to invoke or extend the functionalities of *IC3*, e.g., tool *StoryDroid* [18] enhanced *IC3* to construct storyboard of app, and *IC3-DIALDroid* [16] implemented incremental callback analysis on *IC3* that aims to improve the efficiency and effectiveness. Besides, *Gator* [5] provides an analysis client ATGClient to generate the activity transition graph of an app. However, these works ignore the components connected by fragments or adopt context-insensitive analysis, which will decrease the precision of generated CTG. *FragDroid* [17] presents the activity-fragment transition graph. However, it is not publicly available. Also, its transition analysis only tracks several object-creating APIs while not modeling the ICC behaviors completely.

7 CONCLUSION

Android ICC resolution is widely used in various scenarios. For precise resolution, it faces challenges of fragment analysis and data extraction. In this paper, we present a fragment-aware and context-sensitive ICC resolution tool *ICCBot*, which can efficiently construct CTGs with a higher accuracy and generate friendly outputs.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 62102405), the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036), and Guandong Basic and Applied Basic Research Foundation (Grant no. 2021A1515011562).

REFERENCES

[1] 2015. CSipSimple. https://github.com/r3gis3r/CSipSimple.
[2] 2015. IC3. https://github.com/siis/ic3.
[3] 2017. DroidBench. https://github.com/secure-software-engineering/DroidBench.
[4] 2017. ICC-Bench. https://github.com/fgwei/ICC-Bench.
[5] 2019. GATOR. http://web.cse.ohio-state.edu/presto/software/gator/.
[6] 2020. IC3-DIALDroid. https://github.com/dialdroid-android/ic3-dialdroid.
[7] 2022. F-Droid. https://f-droid.org/.
[8] 2022. Fragment. https://developer.android.com/guide/fragments.
[9] 2022. ICCBot, ICCBotBench. https://github.com/hanada31/ICCBot.
[10] 2022. Intent. https://developer.android.com/guide/components/intents-filters.
[11] 2022. json-handle. https://chrome.google.com/webstore/detail/json-handle/iahnhfdhidomcpggpaimmmahffihkfnj?hl=en.
[12] 2022. Soot. https://github.com/soot-oss/soot.
[13] 2022. Use-define chain. https://en.wikipedia.org/wiki/Use-define_chain.
[14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI 2014*. 29.
[15] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA 2013*. 641–660.
[16] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. 2017. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *AsiaCCS*. 71–85.
[17] Jia Chen, Ge Han, Shanqing Guo, and Wenrui Diao. 2018. FragDroid: Automated User Interface Interaction with Activity and Fragment Analysis in Android Applications. In *DSN 2018*. 398–409.
[18] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. StoryDroid: automated generation of storyboard for Android apps. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 596–607.
[19] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *ICSE 2015*. 77–88.
[20] Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In *USENIX Security Symposium, 2013*. 543–558.