A Comprehensive Evaluation of Android ICC Resolution Techniques

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

Shixin Zhang  
School of Software Engineering  
Beijing Jiaotong University  
Beijing, China  
zhangsx@bjtu.edu.cn

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

Xi Deng  
State Key Lab. of Computer Science  
Institute of Software, CAS, China  
Beijing, China  
dengxi@ios.ac.cn

Jun Yan†  
State Key Lab. of Computer Science  
Institute of Software, CAS, China  
Beijing, China  
yanjun@ios.ac.cn

Jian Zhang†  
State Key Lab. of Computer Science  
Institute of Software, CAS, China  
Beijing, China  
zj@ios.ac.cn

ABSTRACT

Inter-component communication (ICC) is a widely used mechanism in mobile apps, which enables message-based data flow transferring and data passing between Android components. Effective ICC resolution requires precisely identifying entry points, analyzing data values of ICC fields, modeling related framework APIs, etc. Due to various control-flow- and data-flow-related characteristics involved and the lack of oracles for real-world apps, the comprehensive evaluation of ICC resolution techniques is challenging.

To fill this gap, we collect multiple-type benchmark suites with 4,104 apps, covering hand-made apps, open-source, and commercial ones. Considering their differences, various evaluation metrics, e.g., number count, graph structure, and reliable oracle-based metrics, are adopted on-demand. As the oracle for real-world apps is unavailable, we design a dynamic analysis approach to extract the real ICC links triggered during GUI exploration. By auditing the code implementations, we carefully check the extracted ICCs and confirm 1,680 ones to form a reliable oracle set, in which each ICC is labeled with 25 code characteristic tags. The evaluation performed on six state-of-the-art ICC resolution tools shows that 1) the completeness of static ICC resolution results on real-world apps is not satisfactory, as up to 38%-85% ICCs are missed by tools; 2) many wrongly reported ICCs are sent from or received by only a few components and the graph structure information can help the identification; 3) the efficiency of fundamental tools, like ICC resolution ones, should be optimized in both engineering and research aspects. By investigating both the missed and wrongly reported ICCs, we discuss the strengths of different tools for users and summarize eight common FN/FP patterns in ICC resolution for tool developers.

KEYWORDS

Android, Inter-Component Communication (ICC), Transition Graph

ACM Reference Format:


1 INTRODUCTION

Android programs are composed of four types of basic components, which are provided to interact with users, perform background tasks, etc. Each component is a single module and components communicate with each other through the Inter-component communication (ICC) mechanism. To figure out the control and data flows between the source and target components, users can use ICC resolution tools to extract ICC-related information. The most widely used ICC resolution tools are Epice [41] and IC3 [39]. They model the ICC-related framework APIs and perform a data-flow analysis to resolve the ICC field values, whose results can be used to construct the component/activity transition graph (CTG/ATG). Some works [12, 13] use the constructed transition graph to help the program behavior understanding, and others [5, 19, 31] use it to help automatic test generation. Besides, there are various ICC-related vulnerabilities that have attracted the attention of researchers, including inter-component privacy leak [9, 36, 56], permission leak [6, 44, 57], and inter-app collusion [7, 10, 17, 35]. With wide usage in various scenarios, both the soundness and completeness of ICC resolution results have great impact on its applications.

In Android, an ICC message is represented as an Intent [30] object, which contains a set of Intent fields. To obtain the source components of ICCs, we need to find the control flows from entry
point method to the Intent object sending statements, where the entry method may be user-customized ones that are difficult to be identified. And to find out the target component, the values of carried Intent fields should be carefully analyzed. As many code characteristics are involved when identifying the source and target of ICC messages, resolving ICCs with high precision is a challenging task. During analysis, the imprecision in any step, i.e., while handling any code characteristic, may lead to either false positives (FPs) or false negatives (FNs) in the final results. Moreover, these FNs or FPs may be propagated upwards as ICC resolution usually works as a fundamental module, e.g., more than half of FNs in LogExtractor \cite{14} are ICC-related as it invokes the ICC resolution tool IC3 \cite{39}. Actually, for both the users and developers of ICC resolution tools, it is hard to know whether the reported ICCs are trustworthy or not and the root causes of precision loss. Therefore, to figure out that, a comprehensive evaluation focusing on Android ICC resolution techniques is required.

There are several off-the-shelf benchmarks \cite{16, 26} that can be reused for ICC-related evaluation. They are designed by researchers who want to measure tools’ effectiveness when encountering ICC code snippets. Although widely adopted, it is questionable whether these hand-made apps could represent complex real-world ones, for they are designed only with a few code characteristics. For a more practical evaluation based on real-world codes, there are two challenges. Lacking proper metrics is the first challenge. When evaluating apps without available ground truths, many works \cite{10, 39, 41} measure and compare the number of resolved ICC links instead. The number-based comparison is effective only when the tools under evaluation rarely report nonexistent ICCs, i.e., FPs. However, according to the further experimental results on hand-made apps, FPs exist for most ICC resolution tools, which makes the number-based comparison less convincing. Another challenge is the lack of high-quality benchmark suites. A high-quality benchmark suite requires both the representative test inputs, i.e., Android apps, and available test oracle, i.e., ground truth ICCs. To figure out the different behaviors of tools when resolving ICCs with various code characteristics, both the ICC-related code snippets and the involved code characteristics should be identified and labeled for each ICC in the test oracle. Such information can also help developers to find real-world instances for each unhandled characteristic, and give directions for tool updating. However, there are no such characteristics-labeled benchmark suites up to now.

In this paper, we focus on the comprehensive evaluation of widely used ICC resolution tools and pick six state-of-the-art ones as the evaluation subjects. We collect multiple-type benchmark suites with 4,104 apps, including hand-made app sets, large-scale real-world ones, as well as a compact but representative dataset with reliable oracles, with which we can observe tools’ performance on different app sets. For different benchmarks, we adopt different evaluation metrics, in which the number-based metrics have weak credibility but strong versatility, so they fit all the benchmarks; the graph-based metrics require that ICCs are actually designed for real mobile users, so they are suitable for real-world apps but not hand-made ones; and the oracle-based metrics should be applied on datasets with available oracles and code characteristic labels.

For hand-made apps, we can easily label their ICCs as well as the ICC-related code characteristics. As oracles for real-world apps are unavailable, we design a dynamic analysis approach to collect as many real ICC links as possible, because the dynamically triggered ICCs are not limited by the complexity of static code characteristics. First, we adopt the GUI exploration approach to trigger ICCs, which covers 58.9\% app components. By monitoring the execution traces of apps, we propose a specific ICC extraction approach considering the ICC launching characteristics. And to ensure the reliability of collected ICCs, we combine automatic result filters and careful manual code auditing. Finally, we successfully map 1,680 ICCs to corresponding code snippets and label the involved code characteristics, which form a reliable benchmark with ICCs extracted from real-world apps. The apps, ICC oracles and their code characteristic tags are all publicly available here \cite{27, 29}. Through the evaluation, we have the following observations. First, tools behave inconsistently on multiple benchmarks, which reflects the necessity to construct reliable oracles on real-world apps. Especially, the completeness of ICC resolution results on real-world apps is not satisfactory. Up to 38\%-85\% ICCs are missed by tools for their inadequate analysis of specific code characteristics. Second, many fake ICCs are sent from or received by only a few components and number-based metrics could not identify them. With the help of graph-based metrics, we can quickly identify a set of wrongly reported ICCs, which are caused by conservative analysis or the transitivity of imprecision. Besides, most tools suffer from the inefficiency problem when working on complex real-world apps. Users don’t know when the analysis will finish and have to terminate it with no output. Finally, based on the evaluation results, we recommend typical scenarios of tool usage and summarize eight common FN/FP patterns in ICC resolution.

**Contributions.** The contributions of this work are threefold:

- We construct multiple-type benchmark suites for ICC resolution, which contain both hand-made apps designed with specific characteristics and real-world apps with complex ICC implementations, and propose a dynamic ICC extraction approach to obtain characteristic-labeled oracles for representative apps.
- We propose a unified ICC resolution comparison framework and design specific metrics for multiple-type benchmark suites.
- We carry out in-depth evaluations on six popular and state-of-the-art ICC resolution tools, clarify the strengths and weaknesses of each tool, summarize the root causes that lead to precision loss, and discuss the directions for further improvement.

2. ICC RESOLUTION: AN OVERVIEW

This section gives an overview of the ICC mechanism, the state-of-the-art ICC resolution tools and the widely used metrics for ICC evaluation.

2.1 Overview of ICC Mechanism

There are four basic components \cite{15} in Android apps, including Activity, Service, BroadcastReceiver, and ContentProvider. For the convenience of communication among app components, the Android system provides the ICC mechanism. An app component can create an Intent object and send it to the Android system. Along with the Intent, both the basic ICC fields, e.g., action and category, and the user-customized extra data fields will be delivered. The system resolves the value of these fields to infer the
2.2 Existing Tools and Application Scenarios

Researchers have proposed many works that apply the ICC resolution results. One of the most popular application scenarios is security property checking, especially privacy leak detection. In the beginning, the intra-component leak detection [4, 33, 55] is concerned. Considering that many sensitive data are passed by ICC messages, researchers extend the approaches to support inter-component leak detection, including IccTA [36], Amandroid [50] and DroidSafe [22]. Besides privacy leak [9, 56], ICC resolution also relates to permission leak [6, 44, 57], inter-app collusion [7, 8, 10, 17, 35, 38, 48, 58], etc. Another typical scenario are GUI testing, e.g., using the constructed transition model to guide the target-directed test generation [19, 31, 34], and generating the storyboard of apps [12, 13].

With the wide usage of ICC, we start a systemic investigation around ICC resolution from two well-known works, IC3 [39, 40] and Epicc [41]. Among all their citations, we first filter the works without mentioning the tool name explicitly and get 155 citation works upon IC3 and 376 for Epicc. Then we filter the non-English papers, repeated ones, and degree thesis. Papers that just introduce tools as related works are also removed. Totally, we get 48 works that utilize or extend IC3 and 12 papers for Epicc. Six works are found implementing ICC-analysis modules by themselves instead of using IC3/Epicc for efficiency or effectiveness reasons. And five works [10, 12, 32, 45, 52] develop standalone ICC analysis tools. Moreover, we observe that both the analysis framework Gator [54] and an early tool AVE [5] provide ATG analysis functionality.

According to the above investigation, ten state-of-the-art tools are discovered, in which ICCMATT [32] and RAICC [45] are omitted for requiring source code, and only generating refactored application but not ICC links. The rest ones are listed as follows. In 2013, AVE (2013) [5] constructs static ATG and uses it to guide the dynamic test generation. In the same year, Epicc (2013) [41] reduces the discovery of ICC to an instance of the Inter-procedural Distributive Environment (IDE) problem. IC3 (2015) [39] is an enhanced tool based on Epicc, which uses a generic solver to infer possible values of complex objects in an inter-procedural, flow, and context-sensitive manner. GATOR (2015) [54] is a program analysis toolkit that performs static control-flow analysis on Android apps [53]. The ATGClient is one of its default client provided. IC3DIALDroid (2017) [10] (IC3Dial for short) extends IC3 by implementing incremental callback analysis to replace the original one. StoryDroid (2019) [13] aims at generating storyboard for apps, which combines the results provided by IC3 and ICCs extracted with fragments and inner classes features. Another work StoryDistiller (2022) [12] (StoryD for short) is an extension of it, which optimized the original tool on both the ATG construction and UI page rendering. ICCBot (2022) [52] is a code slice and summary-based resolution tool, which considers the modeling of fragment and performs inter-procedural context-sensitive analysis. In the evaluation part, for

$$\text{Algorithm 1: ICC Sending Process}$$

<table>
<thead>
<tr>
<th>Tool</th>
<th>Last Update</th>
<th>APK Input</th>
<th>Graph Output</th>
<th>Functionality</th>
<th>Base Tool/Framework</th>
<th>Approach</th>
<th>Sensitivity</th>
<th>Component Act/NAct/F</th>
<th>Extra Data</th>
<th>StrOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVE [5]</td>
<td>2016-09</td>
<td>✓</td>
<td>✓</td>
<td>ATG*</td>
<td>ATG Construction</td>
<td>SCanDroid/Wala ATG Analysis</td>
<td>✓/ ✓/ ✓/ X</td>
<td>✓/ ✓/ ✓/ X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>ATG-Tool [22]</td>
<td>2020-02</td>
<td>✓</td>
<td>✓</td>
<td>ICC Resolution</td>
<td>ATG/ Soot</td>
<td>IDE Analysis</td>
<td>✓/ ✓/ ✓/ X</td>
<td>✓/ ✓/ ✓/ X</td>
<td>✓</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 1: Overview of the ICC Resolution Tools (✓: True, X: False, -: Unknown, *: To be Discussed)
tools Epicc and StoryDroid, we only adopt their extended version IC3 and StoryDistiller.

Table 1 first gives an overview of the update time, input/output format, functionality of the collected tools. As some tools are developed by extending others, we list their base tool and the fundamental analysis framework. The column approach presents the approaches adopted by each tool, in which Gator uses a simplified IDE analysis of Epicc (according to [53]), and StoryD uses IDE because it first runs IC3 to get parts of ICCs. Then we summarize the analysis sensitivity, including flow, context, field, and object sensitivity [37], of each tool by investigating their related literature. Tools IC3 and IC3Dial both declared that they use context-sensitive inter-procedural analysis, however, we find several context-insensitive counterexamples in the subsequent evaluations. For StoryD, we use the same sensitivity with IC3 because the sensitivity of its own fragment analysis is unknown. The next column gives the analyzed component type, in which A3E and StoryD declared that they only construct ATG, but according to our evaluation results, other kinds of components (NAct) like services and broadcast receivers are also reported. Overall, only StoryD and ICCBot concentrate on the analysis of Fragment (Fr) component. The last two columns give whether there are analyses of extra data and string operation.

2.3 Metrics Adopted by Existing Tools
According to the evaluation approach presented in the related literature, we summarize the number of hand-made (#hm) and real-world (#rw) Android packages (apks) and the evaluation metrics used by each tool. Note that, StoryD has two numbers of #hm and #rw for it has two versions, and A3E is not listed as its transition extraction phase is not directly evaluated. As shown in Table 2, the number of identified ICC links (ICC) and the ratio of the apps that can successfully pass the analysis without timeout or crash (succ) are two popular metrics. Totally, five tools are evaluated with hand-made apps (labeled with *), most of which evaluate the FN and FP ICCs with the labeled ground truths. Six tools are evaluated with real-world apps (labeled with #). However, as no ground truth is available, researchers usually use the number of identified ICC fields that can be determined without uncertainty (if), the ratio of covered activity (cov), and detected leaks in higher-level analysis (leak) as metrics to evaluate the extracted ICCs.

![Figure 2: The Unified Evaluation Framework](image)

Intent matching results; A3E does not filter the ICCs connected with a non-component class; and StoryD does not directly connect the components linked by fragments (Act → Frg → Act). To make the comparison possible, we unify the ICC resolution results with several steps: component filtering and enhancing, target matching, and output unifying. The pre-process of each tool is shown in Fig. 2.

3.2.2 Test Suites for Real-World Apps. As existing works perform evaluations on real-world datasets varying from 50 to 2,000 apps,
we collect 2,000 open source apps from F-droid [18] and 2,000 commercial apps from Google Play [42] as a large-scale benchmark BenchLarge. All the apps are randomly downloaded from the market AndroZoo [3]. However, it is hard to obtain the complete oracle for up to 4,000 apps. Thus, we decide to construct oracles for real-world apps on a compact subset of real-world apps. Considering the representativeness of test suites, we prefer the apps that suit GUI exploration as well as the ones that may have more ICC links. First, we pick all the 20 apps that are used in a recent dynamic exploration work [51], all of which can pass the instrumentation process and are suitable for exploration. One app is dropped for its source code is unavailable by now, so the ICCs of it are difficult to be confirmed. Besides, we consider the downloaded apps in BenchLarge. For the efficiency of manual auditing with source code, only the F-droid apps are taken into consideration. We first analyze the number of components in each app and pick the top 40 apps. Then we filter out the apps that failed the instrumentation and the duplicated ones that are variants of the collected ones. We also drop the social media apps that require a real identity. Finally, we got 31 (19+12) apps (BenchSmall) proper for oracle construction, whose size ranges from 1M to 93M, the average number of GitHub stars is 1,010, and the average number of components is 35. After collection, we have three benchmark suites with 4,104 apps, including a hand-made one, a large-scale real-world one, and a compact real-world app set.

### 3.3 Oracle Construction

For the hand-crafted apps in BenchHand, we perform manual review on code to obtain the ground-truth oracle. However, the specifications for real-world apps in BenchSmall are not available to the third-party testers, which means the ground truths cannot be obtained. An alternative is to manually collect a subset of real ICCs as an under-approximation of the ground truth for evaluation, which is sound but incomplete. To guarantee the usability of the constructed oracle, the oracle obtaining approach should be practical, and the reliability of each ICC should be confirmed. For practical ICC collection, the instrumentation-based dynamic analysis can help to obtain candidate ICCs. After inserting method-level probes into apk files, we can automatically collect the runtime information during GUI exploration, and then analyze the component-launching orders from logs. On one hand, it is applicable for any app that allows apk modification and repackaging. On the other hand, the dynamically triggered ICCs are not limited by the complexity of static code characteristics, i.e., the corresponding code snippets are with high diversity. Since dynamic analysis may not be able to trigger all intents, it would introduce bias to the results. To improve the overall coverage, we combine the results of state-of-the-art GUI input generation tools and manual exploration.

First, we utilize the GUI testing tool APE [23] to drive the dynamic execution. Each app is explored three times and each execution takes one hour. Besides, we manually interact with each app for ten minutes as a supplement. Overall, the dynamic GUI exploration covers 613/1,103 components, with an average coverage of 58.9%.

To guarantee the reliability of oracles, we filter the ICC set with two steps. As the class loading orders could not accurately reflect component transitions, we can not simply use them to build ICCs. Instead, both the lifecycle status of components and the historically visited component stack should be considered. Besides, many lifecycle methods are not overridden by developers, thus the execution of these methods will not be logged. Also, components can extend their father classes and invoke their lifecycle methods, which introduces irrelevant components and messes the event order. Furthermore, there are several types of specific lifecycle behaviors, e.g., launch-mode or flags setting will influence the loading of historically visited components. For these reasons, we adjust our ICC extraction algorithm to fit these problems, including filtering the polymorphic method invocations, omitting the non-starting callbacks of recently launched components, etc. These works can help us to filter parts of FPs and save labor costs in further code auditing. Details of the Android single- and multiple-component interaction models and the ICC extraction process are displayed along with the collected oracle set in [29]. By the automatic dynamic log analysis, we get 1,339 ICCs as the dynamically constructed oracle.

In the next step, we manually filter the ICCs that cannot be linked to an Intent-sending code snippet and confirm the correctness of 984 ICCs. First, we globally search both the name of the target component and values of corresponding intent-filters (declared in the manifest file for implicit ICC), by which we can find the ICC sending methods. Then, we trace their callers with the help of the call graph. If we could find a trace starting from a lifecycle method or a callback method, we can finish the search. Note that, for callback methods, we also need to find out how the callback is registered. If the source component is not registered in the manifest file (may be an abstract class), we then review the code of their subclasses. And because many ICCs are passed through fragments, we will search the loaded fragments in the source code and check whether an Intent is sent by a loaded fragment. After that, if we still cannot find a code snippet, we filter this ICC out of the oracle set. For example, we use the order of callback methods to decide the order of the components, if an ICC is sent by the previously started background services, the source of the ICC may be misidentified. Besides, the dynamically loaded code may trigger real ICCs during runtime, however, these ICCs cannot be recognized by static ICC resolution tools. Other reasons like unmodeled polymorphic relationships and unexpected app restarting also lead to wrongly recognized ICCs during the dynamic analysis. Meanwhile, some ICCs that are hidden behind complex control and data flows may be missed. For example, the longest call trace we successfully tracked involves fourteen method calls and nine classes, including three activities, three fragments, two adapters, etc. It is difficult and time-consuming to confirm ICCs like that. Besides, during locating the related code snippets for the detected ICCs, we also record the newly observed ICCs for oracle enhancement. In this step, 586 ICCs are manually added, after which the number of ICCs is 1,570.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>BenBD</th>
<th>BenI</th>
<th>BenNS</th>
<th>BenR</th>
<th>BenT</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>Fragment</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Callback Entry</td>
<td>0</td>
<td>16</td>
<td>4</td>
<td>0</td>
<td>11</td>
<td>31</td>
</tr>
<tr>
<td>Implicit Match</td>
<td>5</td>
<td>13</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>28</td>
</tr>
<tr>
<td>Calling Context</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>Atypical ICC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>Library</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>DynamicBR</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Str Operation</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Sum</td>
<td>14</td>
<td>40</td>
<td>38</td>
<td>23</td>
<td>33</td>
<td>150</td>
</tr>
</tbody>
</table>
Table 4: Type and Distribution of 25 ICC-related Tags

<table>
<thead>
<tr>
<th>Type</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Activity (96%), Service (10%), Broadcast (5%), Dynamic Broadcast (1%)</td>
</tr>
<tr>
<td>Non-Component</td>
<td>Fragment (14%), Adapter (32%), Widget (4%), Other Class (39%)</td>
</tr>
<tr>
<td>Entry Method</td>
<td>Lifecycle (19%), Dynamic (66%), Implicit (21%), Static (4%)</td>
</tr>
<tr>
<td>Exit Method</td>
<td>Normal (94%), Atypical (6%)</td>
</tr>
<tr>
<td>Method Call</td>
<td>Basic (56%), Callback Listener (53%), Asynchronous (6%)</td>
</tr>
<tr>
<td>Intent Type</td>
<td>Explicit Intent (97%), Implicit Intent (3%)</td>
</tr>
<tr>
<td>Intent Field Value</td>
<td>Context-related (40%), Static Value (1%), Extra Data (40%), String Operation (0.5%)</td>
</tr>
</tbody>
</table>

For each extracted ICC, we review its code snippet and point out the typical characteristics involved. As shown in [29], we design 25 code characteristic tags for each ICC and two of the authors label these tags together. Table 4 gives the type and distribution of these 25 tags, involving the type of source or destination components, the entry method of an ICC invocation, how an ICC is sent out, the details of the method calls and Intent field values.

3.4 Metric Picking

This part discusses the metrics that will be adopted in this study.

Oracle Metric. For apps with ground truths, the oracle-based metrics true positive (TP), false positive (FP), and false negative (FN) are the best choices, i.e., which measure whether an ICC identified by the tools has the same source and destination component name with an ICC in the oracle set. For apps in BenchHand, we can obtain their ground truths by code reviews, and compare their numbers of the TP, FP, and FN ICCs. For the compact BenchSmall, as its under-approximation of the ground truths is extracted, we can get the lower bound of FN ICCs when compared with the labeled oracle.

Number Metric. In existing works, number-based metrics, e.g., the number of reported ICCs and identified ICC fields, are usually used to evaluate the tools’ performance on real-world apps. The reason is that number-based metrics can reflect the upper bound of the TP ICCs, it is useful when there are few FP ICCs. However, according to the oracle-based results on BenchHand, FPs exist for most ICC resolution tools (refer to Table 5). Considering the well versatility on various datasets, we still use number-based metrics to evaluate tools’ performance on all three benchmark suites. Besides, to measure the contribution of the number-based metrics to the ICC resolution results, we take the structure of CTG into consideration, i.e., use the graph-based metrics as a supplement.

Graph Metric. As the results of ICC resolution can be represented as a directed graph, i.e., the nodes are components and the edges are ICCs, we use the average degree metric to obtain the density of edges. \( \text{deg}(\text{CTG}) = 2 \times |E| \div |N| \), in which \(|E|\) is the number of reported ICCs and \(|N|\) is the number of declared components. This metric takes both the number of ICCs and the scale of apps into consideration. Larger \( \text{deg}(\text{CTG}) \) means more ICCs reported, which can be used to make comparisons among multiple benchmarks. Besides, we consider the connectivity of the graph with the following three metrics. The metric \( C_{\text{separated}} \) denotes the number of isolated components that do not connect to any other; \( C_{\text{mainNot}} \) denotes the number of components that are not reachable from the default entry, usually the MainActivity; and \( C_{\text{exportNot}} \) denotes the number of components that are not reachable from any exported entry component. From the users’ perspective, the lower these metrics, the better CTG connectivity, and more functionalities could be explored. There are two possible cases that a component may not be linked to any other component. One case is, it is an exported component only for external launch. In our dataset, most components (85%) are either MainActivity or not exported, which should connect to others. Meanwhile, many exported activities are not designed for external launch only, i.e., though they can be launched externally, they can also be launched internally, e.g., payment or login activities. The other case is dead-code components that are registered but not used, which also rarely happens. Therefore, we suppose that developers are less likely to design separated or unreachable components in their apps on purpose, and the graph-based metric can work on most scenarios.

4 RESULTS AND ANALYSES

This section aims to answer the following research questions.

- RQ1: Can existing tools analyze multiple-type apps with high success rate and efficiency?
- RQ2: How is the performance of the tools in terms of number & graph metrics?
- RQ3: To what extent can the tools identify ICCs in our oracle set?

4.1 RQ1: Usability and Efficiency

First, we explore the configurability of tools. For the most popular tool IC3, there are seven COAL [39] models that can be configured in its source code. However, it brings the extra cost for users to learn the principle and grammar of COAL. Though both IC3Dial and StoryD extend IC3, i.e., are IC3-based tools, they do not modify the inner models. As IC3Dial optimizes the callback related code snippets and StoryD directly invokes IC3, they have no extra configuration item. For StoryD, we remove the code related to the dynamic app exploration process and only record the statically extracted ICCs. Tools Gator and ICCBot both provide various configuration items. By inspecting the argument parsing process of Gator, we find the “implicitIntent” item relates to ICC resolution and set it as true. For ICCBot, we use all its default configurations.

Then, we compare the success rate during analysis and the execution time of tools. The whole analysis process is performed on a Linux server with two Intel® Xeon® E5-2680 v4 CPUs and 256 GB of memory. As shown in Fig. 3, on BenchHand, all apps can be successfully analyzed within an acceptable time. On the real-world app set BenchSmall, Gator outperforms others in efficiency while IC3-based tools are more time-consuming, e.g., IC3 takes more than six hours to analyze app SuntimesWidget. As the users usually invoke fundamental tools in limited time, e.g., 30 minutes in [2], we use the same setting when analyzing dataset BenchLarge. With such a limit, IC3 and IC3Dial suffer from crash or timeout problems and have a lower success rate than others, e.g., they cannot finish analysis for up to 36% and 17% google play apps. For StoryD, though it invokes IC3, it benefits from the 10-minute timeout setting on invoking IC3 and light-weight self enhancement. The different success rates of F-droid and Google Play apps on BenchLarge also indicate that the complexity of code can greatly influence the results, and the analyzing efficiency on complex real-world apps requires more attention. In total, the analysis time for all tools on BenchHand, BenchSmall and BenchLarge are 2, 13 and 892 hours, respectively.
4.2 RQ2: Use Number & Graph Metrics

In Fig. 4, we count the number of edges that involve the basic component only (C-C), the activity component only (A-A), and both the basic component and fragment (CF-CF) on each benchmark. Dataset BenchLarge is separated into two subsets: F-droid and Google Play set. As we can see, the behaviors on BenchHand are different from the others, e.g., IC3Dial generates more ICCs on BenchHand while generating fewer ICCs on the other datasets, and the result of Gator is the opposite. The reason is that hand-made apps usually cover the basic usages of one code feature or the combination of a set of features. But it is difficult to design specific code snippets that can cover the FN/FP-related complex patterns that occur in real-world code. Thus, BenchHand is useful to evaluate tools’ effectiveness on specific characteristics, while the results are not representative enough due to the differences in code features between hand-made and real-world apps.

On all datasets with real-world apps, Gator reported the most ICC edges. Especially, on the Google Play dataset, it generates more than 300,000 edges, which is 4-80 times more than all others. Unfortunately, those results are confusing because we do not know whether they are caused by better analysis ability or higher FP rates. To figure out that question, we evaluate tools with the graph-based metrics deg(CTG), Cseparated, CmainNot and CexportNot on the constructed CTGs. The average values of these metrics on three datasets are displayed in Fig. 5, in which the left Y-axis is for deg (CTG) and the right Y-axis is for others. Along with a large number of ICCs reported, Gator also has high degree values. However, its connectivity-related values are similar to tools that report much fewer ICCs than it, which means many newly added ICCs do not contribute to connectivity improvement. This abnormal behavior guides us to identify many FP ICC candidates in the following Section 5.2. Compared to it, tools that have both a relatively high degree and high graph connectivity are more reasonable. In summary, the structure-related information, e.g., the graph-based metrics, can help users notice the unusual behaviors.

4.3 RQ3: Use Oracle Metrics

Both the number and graph metrics can give us an overview of the ICC resolution results. To obtain more reliable evaluation results, we then measure the effectiveness of tools with oracle metrics.

4.3.1 On BenchHand. With the labeled oracles, Table 5 gives the evaluation results on benchmark BenchHand. The second column gives the number of ICCs in the oracle set (OR) of each benchmark, and the other columns give both the number of FP and FN ICCs. It shows that FPs happen less often than FNs on most benchmarks, except BenT, on which tool IC3Dial generates many FPs while IC3 and Gator also generate a few FPs. By reviewing these FP-related reports, we find that the reachability analysis of methods and the context value tracking results affect the results. For the reachability computing, some tools first compute the methods that could reach an Intent-sending statement, and then compute the data values that could be assigned to that Intent object. During the two-step reachability computing, the relationship between these context values and the method call is omitted. For example, in the case study in Figure 7, the decorator method may be reused by multiple callers under various contexts, which leads to many FPs by tools. Many ICCs are missed on BenchSmall as it contains various atypical types of ICC usage, which requires the modeling of specific APIs. BenT also leads to many FNs because it uses several not commonly used callback methods. Overall, on BenchHand, StoryD and A3E behave well on the FP rate, ICCBot behaves well on both FP and FN rates.

<table>
<thead>
<tr>
<th>Bench</th>
<th>#OR</th>
<th>Gator</th>
<th>IC3</th>
<th>IC3Dial</th>
<th>A3E</th>
<th>StoryD</th>
<th>ICCBot</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenchA</td>
<td>12</td>
<td>0/1</td>
<td>0/4</td>
<td>0/4</td>
<td>0/9</td>
<td>0/9</td>
<td>0/2</td>
</tr>
<tr>
<td>BenchB</td>
<td>26</td>
<td>0/22</td>
<td>0/16</td>
<td>0/3</td>
<td>0/19</td>
<td>0/19</td>
<td>0/0</td>
</tr>
<tr>
<td>BenchC</td>
<td>37</td>
<td>0/4</td>
<td>0/4</td>
<td>0/4</td>
<td>0/37</td>
<td>0/1</td>
<td>0/0</td>
</tr>
<tr>
<td>BenchD</td>
<td>24</td>
<td>0/24</td>
<td>0/23</td>
<td>0/23</td>
<td>0/1</td>
<td>0/24</td>
<td>1/0</td>
</tr>
<tr>
<td>BenchE</td>
<td>11</td>
<td>3/4</td>
<td>3/4</td>
<td>24/1</td>
<td>0/12</td>
<td>0/10</td>
<td>0/0</td>
</tr>
<tr>
<td>Sum</td>
<td>110</td>
<td>4/59</td>
<td>3/51</td>
<td>24/35</td>
<td>1/78</td>
<td>0/63</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Table 5: Evaluating with Oracle on BenchHand
Based on the labeled information, we further compare the distribution of code characteristics of FN ICCs. Compared with other tools, both ICCBot and $A^3E$ work well with characteristic *atypical ICC*, as ICCBot adds atypical APIs into the Intent model while $A^3E$ simply reports ICCs if the creation of an Intent object is identified. Both $A^3E$ and StoryD fail to identify ICCs with implicit Intent. According to Fig. 2, they both generate CTG directly but do not apply implicit matching, while others consider it by themselves or by the target matching module in our unified framework. Thus, though StoryD invokes IC3, it has more FNs than IC3 on some cases. Compared to others, $A^3E$ is the only tool that failed to resolve all the calling context related ICCs.

4.3.2 On BenchSmall. Fig. 6 gives the hot-map graph of FN rate results on BenchSmall, in which each unit square denotes the FN rate on one app, and X-axis displays the 31 apps that are sorted by the number of ICCs in the oracle set. Using reliable oracles with 1,570 edges, we find that around 38% to 85% ICCs are missed by the six picked tools, and their average FN rates on apps vary from 21% to 88%. That is to say, there are still massive FN ICCs when working on real-world apps. Then, we perform pairwise comparisons to figure out the common and unique ICCs reported by tools. In Fig. 7, the bottom left figures are about all the reported ICCs, whose Y-axis values are the ratios of reported ICCs in the union of ICCs reported by two tools. And the upper right ones only count the TP ICCs, whose Y-axis values are the ratios of TP ICCs in the oracle ICC set. As we can see, IC3 covers all the reported and the TP ICCs of IC3Dial on this benchmark, and the results of other tools are all overlapped. For instance, even though ICCBot can cover most TPs reported by others, every other tool can still report a few TP ICCs that are missed by it. Furthermore, the union of any two tools cannot cover all ICCs in the oracle set.

Fig. 8 presents the top FN-related characteristics of each tool on BenchSmall, in which the left bar shows the characteristics of all the ICCs in the oracle set, and others are about the FN ICCs of each tool. Note again that one ICC may have multiple characteristic labels and the failed apps are not counted for each tool. According to the results, the callback entry related, especially the dynamic and implicit callbacks (CB), ICCs and FNs are both on a large scale. One reason is, callback entry identification is a big challenge due to the various forms of entry declaration. Moreover, many other characteristics show up together with callback characteristics, so they may be repeatedly counted. Compared with the results on hand-made apps, the non-basic-component related characteristics are popular, including the use of *fragment*, *adapter*, etc., which means that the ICC sending procedure in real-world code is much more complex than in hand-made snippets. The Java-specific characteristics *polymorphic* and *asynchronous* have great influence on the method control flow. Many ICCs related to them failed to be extracted. And characteristics like *string operation* and *dynamic broadcast receiver* are not counted because few FNs relate to them, in which string manipulations are more often used in malicious apps but only benign apps are picked in our study.

To avoid the mutual influences among characteristics, we take each characteristic as a separated control variable and count the ICCs that are only related to it. For callback-related ones, we pick ICCs that only satisfy one callback type but are not labeled with any other characteristics. For other characteristics, we omit their callback setting because most ICCs relate to callbacks. As the atypical ICCs are usually related to non-activity components, their component types are not limited. The FN and TP results of each tool are shown in Fig. 9. As we can see, the Java-specific characteristic *asynchronous* is only concerned by two tools, ICCBot and StoryD, and characteristic *polymorphic* is omitted by tool $A^3E$. Though static callback is not a new Android feature, IC3, IC3Dial and $A^3E$ all fail with the characteristic. Among all characteristics, three have tight relationships with the evolution of the Android framework, including *implicit callback*, *fragment*, and *atypical ICC*, which are called newly-introduced characteristics, and others are pre-existing ones. By evaluating the number of ICCs influenced by each single characteristic, we find that more missed ICCs are caused by the inadequate analysis of pre-existing characteristics (73%) than the newly-introduced ones (27%). So, even without consideration of the evolution of the Android framework, there is still a long way to go to improve the precision of ICC resolution tools.

4.4 Observations for Further Improvement

According to the above evaluation results, we have the following observations worthy to be discussed.

*(For Tool Developer)* First, the standard evaluation benchmark suites and suitable metrics for fundamental analysis modules are required. As we can see, there are great differences between self-made and complex commercial codes. For further tools that work on ICC resolution, developers could reuse the datasets and metrics provided in this paper. For other problems, developers...
can leverage the dynamic information to help the benchmark construction for static tools, and vice versa. Moreover, as the oracles on the complex dataset are not available, it may be helpful to utilize the structure-properties of results, e.g., consider the design intention of CTGs when evaluating ICC transitions, in evaluation.

(For Tool Developer) Second, the efficiency of static analyzers should raise more attention. Efficiency and efficiency-induced execution failures usually trouble users [11, 49]. According to our evaluation, the efficiency on complex real-world apps is not satisfactory for most tools, as some expensive analysis approaches may bring unpredictable time costs for users and may not bring equivalent benefits. A simple but practical strategy is to store the intermediate results during analysis and allow users to see the partial results at a certain time point. Besides, how to effectively distribute computation resources during analysis should be further explored. For example, developers can make a quick scan of code to decide the order of analysis units, e.g., class or method. Or they can dynamically evaluate the time cost of each analysis unit and handle the costly ones specifically.

(For Tool User) Third, concerning more about the trustworthy analysis chain. Many high-level analyses rely on the ICC resolution results, and ICC resolution also relies on the precision of other modules. As the imprecision in the low-level analysis may be propagated to the higher level, the imprecision in low-level tools may greatly influence the final performance with unclear root causes. Thus, users should get a comprehensive look at any invoked tools to build a trustworthy analysis chain. And more experimental research should be performed to give a many-sided overview of various fundamental analysis tools.

(For Tool User) Forth, keeping aware of your key requirements. According to the results, numerous ICCs are missed by six state-of-the-art tools, which means there is not a perfect solution to resolve ICCs from complex real-world apps. Therefore, based on our comprehensive evaluation results, users should make decisions according to their key requirements. Here, we give a group of possible requirements and the corresponding candidate tools in Table 6, involving the efficiency, completeness, soundness, scenes to be used, and key characteristics concerned. For example, ICCBot works well in terms of both efficiency and effectiveness and supports all types of components, which can be directly used for CTG construction. Gator outperforms other tools in analysis efficiency so that it can be used in time-conscious scenarios. For the updating of IC3-based tools, StoryD and IC3Dial can be adopted as they concern fragment and entry-point identification, respectively. Note that IC3Dial only works well on parts of apps and the reason is unclear. StoryD provides dynamic UI reference and Gator has a static UI analysis client, so they can combine with layout analysis. And if users pay attention to the data carried with ICC, they can try ICCBot, IC3, Gator, which have such Intent-field analysis.

Table 6: Candidate Tools with Key Requirements

<table>
<thead>
<tr>
<th>Requirement &amp; Candidate Tools</th>
<th>Requirement &amp; Candidate Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less time cost → Gator, A3E, ICCBot</td>
<td>IC3-based update → StoryD, IC3Dial, IC3</td>
</tr>
<tr>
<td>More real ICCs → ICCBot, Gator, StoryD</td>
<td>Ul analysis → StoryD, Gator</td>
</tr>
<tr>
<td>Fewer fake ICCs → ICCBot, StoryD, IC3</td>
<td>Intent field extract → ICCBot, IC3, Gator</td>
</tr>
<tr>
<td>High SuccRate → ICCBot, StoryD, A3E</td>
<td>Fragment-aware ← ICCBot, StoryD</td>
</tr>
</tbody>
</table>

5 ROOT CAUSES AND PATTERNS

5.1 False Negatives in ICC Resolution

As shown in the pairwise comparison results between tools, tools have their specific FN ICC sets. By comparing their differences, we separate FN ICCs into two categories: missed by parts of tools and missed by all tools. For the ICCs missed by parts of tools, one reason is the lack of analysis on specific characteristics, e.g., the omission of fragment by Gator. Meanwhile, the efficiency of the analysis approach is another reason. Many ICCs are missed because some tools cannot finish the analysis within a given time, e.g., IC3 reaches timeout on many apps. By comparing the FN set of tools, we find that 158 ICCs are missed by all tools. Then we analyze the value of the 25 labeled tags of these 158 ICCs, compare the distribution of tags on these ICCs and on all ICCs (refer to Table 4), and find several tags have a higher ratio on the commonly missed ICCs, including fragment, static callback, etc. Based on the ICC triggering path labeled in our dataset, two of the authors discuss how can a specific tag characteristic influence the identification of ICC. Finally, we find 26 ICCs are layout-related, 75 involve multiple callbacks, 49 for inter-procedural assignment, 26 are about container-modeling, and we also find 5 special cases caused by implicit assignment and record it. Five common FN patterns are discussed as follows.

P1: Layout-related Callback. There are several forms of callback entries related to XML layout files. The first line in Listing 1 gives the normal type of static callback, which declares a callback for a button widget. Following, a customized view navView is statically declared, which indeed has a dynamic callback in navView.class and will send ICC in this callback method. Besides, the PreferenceScreen provided by the Android framework
supports another implicit way to trigger Intents. We list one of its usage here. All these patterns require the analysis of layout files.

**Listing 1: FN: Layout-related Callback**

```
<com.pkg.navView class="NavigationItemSelectedListener">
<OnNavigationItemSelectedListener> public void onNavigationItemSelected(View view) {
  startActivity(new Intent(this, Tgt.class));
  };
</Preference>
```

**P2: Multi-step Callback.** In some cases, the callback recognition requires multiple analyzing steps. This type of design is commonly used, e.g., reach a view that may trigger Intent sending after clicking another widget. In Listing 2, the callback `onDrawerOpened()` is hidden behind the callback `onClick()` and the asynchronous method `run()`. This pattern requires precise graph construction as well as a multiple-turn callback analysis.

**Listing 2: FN: Multi-step Callback**

```
public void onCreate(){
  // In Component A.class (A to Tgt)
  public void run() {
    addListener(new DrawerToggle());
    listeners.add(onDrawerOpened(View v){
      startActivity(new Intent(this, Tgt.class));
    });
  }
  public void onClick(View v) {
    // In com.pkg.navView.class
    new Handler().post(pendingRunnable);
  }
  // In com.pkg.navView.class
  setNavigationViewItemSelectedListener(new OnNavigationItemSelectedListener(){
    public void onNavigationItemSelected(View v) {
      startActivity(new Intent(this, Tgt.class));
    };
  };
  // In Component B.class and its layout file (B to Tgt)
  public void onCreate(Bundle savedInstanceState) {
    addPreferencesFromResource(R.xml.item); }
  <Preference android:key="target"> //one Preference in the PreferenceScreen
  <Preference android:key="target"> // In Fragment F.class
  <Preference android:key="target"> // In Fragment F.class
  <Preference android:key="target"> // In Fragment F.class
  <Preference android:key="target"> // In Fragment F.class
```

**P3: Inter-procedural Assignment.** In Listing 3, the Intent object is obtained from the return value of method `getIntentObj()`. For inter-procedural assignments, the passed value can be parameters, return values, and even the static variables. Note that, without tracking a global path, it is difficult to get the precise value of static variables. For others, careful inter-procedural analysis is required.

**Listing 3: FN: Inter-procedural Assignment**

```
Listing 3: FN: Inter-procedural Assignment

// In Component A.class (A to Tgt)
public void onCreate(){
  startActivity(B.getIntentObj(A.this)); }
// In Component B.class
public static Intent getIntentObj(Context ctx){
  return new Intent(ctx, Tgt.class); }
```

**P4: Container Modeling.** Adapter is a widely used data container that is not well-modeled by now. Not only the constant data can be stored in adapters, but fragment instances can also be added to it. The combination of adapter and fragment is popular when using `ViewPager` component, which is used to switch views according to user operation and each view can be a fragment contained in the adapter. Like Listing 4 shows, component `A` loads fragment `F`, whose instance is stored in `mAdapter`. And the fragment `F` launches component `Tgt` when attached. Without the modeling of adapter operating APIs, we cannot figure out which fragment is loaded here. Besides multiple types of adapters, there are also various types of data containers, whose modeling is a challenge to both the control flow edge and data value extraction.

**Listing 4: FN: Container Modeling**

```
Listing 4: FN: Container Modeling

// In Component A.class (A to Tgt)
public void onCreate(){
  nAdapter = new FragmentPagerAdapter(getActivity().getSupportFragmentManager());
  mPagerAdapter.setAdapter(nAdapter);
  nAdapter = new FragmentPagerAdapter(getActivity().getSupportFragmentManager());
  mPagerAdapter.setAdapter(nAdapter);
  public void getItem(int position) {
     switch (position) {
          case 0: return new F(); ...
        }
  }
  // In Fragment F.class
  public void onAttach(Activity act) {
    ((OnDoActionListener) act).onDoAction();
  }
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Fragment F = getSupportFragmentManager().findFragmentByTag("Tgt");
    startActivity(new Intent(F, Tgt.class));
  }
  public void onDoAction() {
    launchActivity(new Intent(this, Tgt.class));
  }
  // In Fragment F.class
  public void onAttach(Activity act) {
    ((OnDoActionListener) act).onDoAction();
  }
```

**P5: Implicit Assignment.** In Listing 5, component `A` first loads fragment `F`. As the fragment `F` is attached, it invokes the method `onDoAction()` in component `A`, which triggers the ICC `A → Tgt`. However, to detect it, we have to know the actual value of the parameter `Act` in the Android framework, the parameter of `onAttach()` equals the host Activity of the current fragment, which is an implicit data assignment. Thus, besides the modeling of the fragment loading behaviors, it also requires modeling the implicit data relationships like this.

**Listing 5: FN: Implicit Assignment**

```
Listing 5: FN: Implicit Assignment

// In Component A.class (A to Tgt)
public void onCreate(){
  loadFragment(F.class);
  public void onDoAction(){
    launchActivity(new Intent(this, Tgt.class));
  }
  // In Fragment F.class
  public void onAttach(Activity act) {
    ((OnDoActionListener) act).onDoAction();
  }
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Fragment F = getSupportFragmentManager().findFragmentByTag("Tgt");
    startActivity(new Intent(F, Tgt.class));
```

### 5.2 False Positives in ICC Resolution

To find out the possible FPs, we pick up apps with the highest value of deg(CTG) for investigation (deg(CTG)>15), including `OpenKeychain` (Gator, 19.0), `easydiary` (Gator, 25.4), `SuntimesWidget` (ICS, 22.5), etc. For these apps, we carefully read their code and infer why a non-existent ICC is reported. The process is the same as how we identified the correctness of ICC during the dynamic analysis (refer to Section 3). After that process, there are still some ICCs failed to be confirmed. For these cases, we try to infer why a nonexistent ICC is reported as tools do not provide details about why they report such an ICC. For the possible patterns, we also construct test cases to verify whether an inferred FP pattern can indeed lead to FPs or not. The final three patterns we observed (P6-P8) are all verified. Through experiments, we also find the simplified model will lead to FPs. For example, `A3E` only identifies the Intent declaration statements but not the complete behavior of Intents, so that fake Intents that are not really sent out are reported. ICCBot tries to track the entry points of ICC. For the complex callback registrations that are missed, it takes the top method that it could track as the entry method, while sometimes this simplification brings errors. Finally, we summarize three concrete circumstances that lead to FPs.

**P6: Polymorphic Invocation.** The invocation relationships become complex when encountering the `polymorphic` characteristic. In Listing 6, subclasses `SonA`, `SonB` and `SonC` all extend class `Father` and implement the abstract method `fatherMethod()`, which is invoked in method `onCreate()`. Obviously, there are two ICCs, i.e., `SonA` launches `SonB`, and `SonB` launches `Tgt`. However, `IC3` reports four ICCs, and `Gator` reports six ones. They both compute the reachability between the Intent sending statements and basic component classes, in which the reachability depends on the precision of the call graph. When combined with the `polymorphic` characteristic, the invocation of a method depends on the execution context, the omitting of which will wrongly connect methods and raise incorrect ICCs. In this example, tools take all the implementations of `fatherMethod()` in the same way, which leads to fake call edges. Moreover, this problem is unexpectedly expanded for `Gator`. In
SonB, method `getIntent()` is invoked to receive Intent from outside. Without object-sensitive analysis, `Gator` misidentifies two Intent objects and takes all the possible sources of `SonB` as the source ICC being sent out, including the FP sources SonB and SonC. The transitivity of FPs may cause exponential growth of ICC numbers.

**Listing 6: FP: Polymorphic Invocation**

```java
public static void launchAct(Context ctx, Intent i) {
    // In Class Util.class
    startActivity(i); ctx.startActivity(i);
}
```

**P7: Decorator Method.** In Listing 7, method `launchAct()` is a decorator method that invokes the API `startActivity()` and adds Intent flags for it. Both components A and B invoke the method `launchAct()` and pass an Intent object to it. However, both the `IC3`-based tools and `Gator` adopt context-insensitive analysis for decorator methods, which means the possible targets for `launchAct()` are extracted from all the received Intents and the sources are all the caller components. In this case, components A and B are the sources, C and D are the targets, i.e., all the four ICCs will be reported, in which two of them (A → D, B → C) are FPs.

**Listing 7: FP: Decorator Method**

```java
// In Component A.class
public void onCreate(){ Util.launchAct(new Intent(getBaseContext(), D.class));}
// In Component B.class
public void onCreate(){ Util.launchAct(new Intent(getBaseContext(), D.class));}
```

**P8: Type Inference.** In Listing 8, component A dynamically registers two broadcast receivers and sets corresponding intent-filters for them. Then, it sends a broadcast with the action value "FilterA", which should be received by the instance br1 of class Receiver1. However, in `IC3`, both Receiver1 and Receiver2 are labeled as the receiving target classes. By debugging, we find that `IC3` failed to track the correct type of br1 and br2 for they are field variables. By a conservative analysis, it takes all the broadcast receivers in the app as the target types for registration, i.e., the two intent-filters are registered to both receiver types. Without carefully concentrating on the scope of variables and the type inference, FP ICCs can be wrongly reported.

**Listing 8: FP: Type Inference**

```java
public class A extends Activity { // In Component A.class
    BroadcastReceiver br1, br2;
    public void onCreate(Bundle savedInstanceState) { br1 = new Receiver1(); br2 = new Receiver2(); registerReceiver(br1, new IntentFilter("FilterA"); registerReceiver(br2, new IntentFilter("FilterB"); sendBroadcast(new Intent("FilterA")); ...
```


