A Comprehensive Evaluation of Android ICC Resolution Techniques

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
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
Univ. of Chinese Academy of Sciences
Beijing, China
dengxi@ios.ac.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

Inter-component communication (ICC) is a widely used mechanism in mobile apps, which enables message-based control 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:
Jiwei Yan, Shixin Zhang, Yepang Liu, Xi Deng, Jun Yan, and Jian Zhang. 2022. A Comprehensive Evaluation of Android ICC Resolution Techniques. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3560420

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 *Epicc* [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

*Yepang Liu is affiliated with the Research Institute of Trustworthy Autonomous Systems, Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, and Department of Computer Science and Engineering of SUSTech.
†Corresponding Authors.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3560420

point methods 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* [14] are ICC-related as it invokes the ICC resolution tool *IC3* [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 [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 [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 [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 implementation, 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 [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 fileds to infer the

Table 1: Overview of the ICC Resolution Tools (✓: True, X: False, -: Unknown, *: To be Discussed)

Tool	Last Update	Apk input	Graph Output	Functionality	Base Tool/ Framework	Approach	Sensitivity F/ C/ I/ O	Component Act/NAct/Fr	Extra Data	StrOp
A^{3}E [1]	2016-09	✓	ATG*	ATG Construction	SCanDroid/ Wala	Taint Analysis	-	✓/ ✓*/ X	X	-
IC3 [24]	2015-09	✓	X	ICC Resolution	Epicc/ Soot	IDE Analysis	✓/ ✓*/ ✓/ X	✓/ ✓/ X	✓	✓
IC3$_{Dial}$ [25]	2020-02	✓	X	ICC Resolution	IC3/ Soot	IDE Analysis	✓/ ✓*/ ✓/ X	✓/ ✓/ X	✓	✓
Gator [21]	2019-05	✓	ATG	ATG Construction	/Soot	IDE Analysis*	X/ X/ -/ -	✓/ X / X	X	X
StoryD [46]	2022-03	✓	ATG*	Storyboard Generation	IC3/ Soot	IDE Analysis*	(✓/ ✓/ ✓/ X)*	✓/ ✓*/ ✓	X	✓
ICCBot [28]	2022-04	✓	CTG	ICC Resolution	/Soot	Slice, Summary	✓/ ✓/ ✓/ ✓	✓/ ✓/ ✓	✓	✓

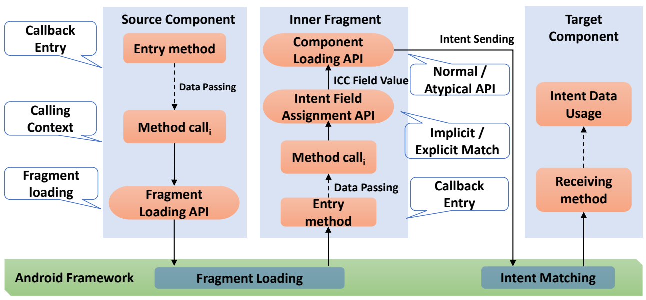

Figure 1: Overview of ICC Sending Process

target components. Fig. 1 displays the overview of the ICC sending process with five commonly used characteristics in ICC resolution.

Callback Entry. Unlike Java programs, there is no main method in an Android applications. Instead, each component has a set of entry methods, including lifecycle methods (e.g., `onCreate()`) and callbacks (e.g., `onClick()`). Both of them are invoked by the Android system in response to GUI or system events. The callback recognition is challenging, for callbacks could be registered in Android framework classes or libraries (implicit callback), registered in code of application package (dynamic callback), or declared in the XML files (static callback). And the ICC invocation may be hidden behind a series of complex callback triggering.

Calling Context. The data extraction of the Intent object fields is essential to ICC resolution. Considering both the Intent object itself and the data of Intent fields can be passed among function calls, a context-sensitive inter-procedural analysis should be performed.

Fragment. Fragment is a dynamically loaded building block of an app’s user interface, which is hosted by an activity or another fragment [20]. In Fig. 1, the source component first loads an inner fragment `f`, i.e., invokes its callback `c`. Then, `f` sends an `Intent` out in one of its methods, which is reachable from `c`. In this case, without fragment modeling, the entry point tracking analysis may terminate at the lifecycle method of a fragment, but miss the actual entry method in its host activity.

Implicit Match. There are two types of Intent. For explicit Intent, their target components can be obtained by analyzing the value of the class or component name related fields. For implicit Intent, the values of fields, e.g., action and category, that are related to implicit matching will determine the destination class.

Atypical ICC. For each ICC, it will be delivered to the Android system through specific API, i.e., the exit point. Besides normally used ICC sending APIs, like `startActivity()`, there are many atypical ICC-related APIs [45] in the Android framework, which also work as exit points although the official Android documentation does not specifically discuss them, e.g., `sendIntent()`. These atypical ICCs can also establish ICC links and should be concerned during ICC resolution.

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 A^3E [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, **A^{3}E (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] (IC3$_{Dial}$ 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 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 **f**low, **c**ontext, f**i**eld, and **o**bject sensitivity [37], of each tool by investigating their related literature. Tools *IC3* and $IC3_{Dial}$ both declared that they use context-sensitive interprocedural 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 A^3E 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 A^3E 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.

Table 2: Evaluation Metrics Adopted By Tools

Tool	#hm	#rw	ICC	succ	FP/FN	if	cov	leak
Epicc	0	1,200				★		
IC3	0	500	★			★		
$IC3_{Dial}$	190	1,000	● ★	● ★				
Gator	20	0	●		●			
StoryD	10/0	50/150	● ★		●		★	
RAICC	20	1,000	★		●			★
ICCBot	132	2,000	●★	●★	●			

3 EXPERIMENTAL SETUP: DATA & METRICS

This section presents the experimental setup, including the unified framework, data collection process, and metric picking principles.

3.1 Unified Evaluation Framework

Before evaluation, we noticed that the functionalities and output formats of the state-of-the-art tools vary. For example, tools *IC3* and $IC3_{Dial}$ only give the data value of ICC fields instead of the Intent matching results; A^3E 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.

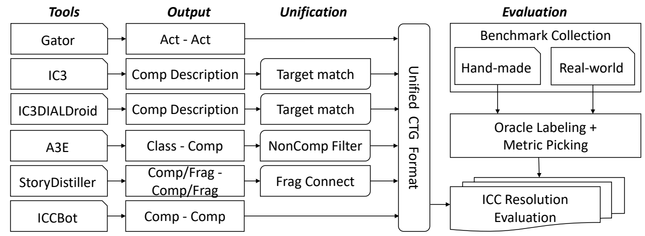

Figure 2: The Unified Evaluation Framework

3.2 Multiple-type Benchmark Collection

In this part, we introduce the collection of multiple-type benchmark suites, including hand-made and real-world apps.

3.2.1 Test Suites for Hand-made Apps. For hand-made benchmarks, we prefer the ones used in existing evaluation approaches, which are carefully designed with specific characteristics. By reviewing the literature that have benchmarks proposed, five hand-made benchmarks with 73 apps are collected as **BenchHand**. DroidBench **(BenD)** [16] in paper [4] is a benchmark suite that is designed for evaluating the information-flow analyzers. It contains 18 ICC-related apps in its ICC category folder. The benchmark ICC-Bench **(BenI)** [26] in literature [50] is an ICC-specific benchmark, which contains 24 apps for testing ICC resolution capabilities. The StoryD-Bench **(BenS)** [47] in literature [13] concentrates on two ICC-related characteristics, fragment and inner class, that are not well-addressed by tool *IC3*. It designs 10 test apps around them specifically. As previous benchmarks do not take the atypical ICC into account, work [45] designed 20 test apps as RAICC-Bench **(BenR)** [43] to supplement *BenD*, in which various atypical ICC usages are provided. And ICCBotBench **(BenT)** [28] in paper [52] is a compact benchmark with one app, which considers various typical usage of fragment loading and data passing among methods.

The ICC-related key characteristics used in these benchmarks can be categorized into several classes, including the control-flow-related characteristics: *callback entry* and *fragment*; data-flow-related ones: *calling context* and *string operation*; ICC-behavior-related: *implicit match*, *atypical ICC* and *dynamic broadcast receiver*; as well as other-class-related: *library*. An ICC that does not involve any specific characteristic is labeled as *basic*. Among them, five characteristics have been introduced in Section 2.1. For others, they focus on whether the assignments of ICC resolution related fields are operated by String APIs (*Str Operation*), whether the modeling of specific Android, Java or the third-party library classes is required (*Library*), as well as whether the target broadcast receiver component is declared dynamically (*DynamicBR*). Table 3 presents the number of related code snippets of each characteristic, in which one code snippet may involve multiple characteristics.

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,

Table 3: Characteristic-Specific Code in *BenchHand*

Characteristic	BenD	BenI	BenS	BenR	BenT	Sum
Basic	2	8	15	1	0	26
Fragment	0	0	4	0	4	8
Callback Entry	0	16	4	0	11	31
Implicit Match	5	13	0	1	9	28
Calling Context	0	0	15	0	9	24
Atypical ICC	0	0	0	23	0	23
Library	5	0	0	0	0	5
DynamicBR	1	2	0	0	0	3
Str Operation	1	1	0	0	0	2
Sum	**14**	**40**	**38**	**25**	**33**	**150**

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 can not 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 component 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 4: Type and Distribution of 25 ICC-related Tags

Type	Distribution
Component	Activity (96%), Service (10%), Broadcast (5%), Dynamic Broadcast (1%)
Non-Component	Fragment (14%), Adapter (32%), Widget (4%), Other Class (39%)
Entry Method	Lifecycle (79%), Dynamic (60%), Implicit (51%), Static (4%)
Exit Method	Normal (94%), Atypical (6%)
Method Call	Basic (56%), Callback Listener (53%), Asynchronous (6%), Polymorphic (42%), Library Method (7%)
Intent Type	Explicit Intent (97%), Implicit Intent (3%)
Intent Field Value	Context-related (40%), Static Value (1%), Extra Data (40%), String Operation (0.5%)

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. $deg(CTG) = 2 \times |E| \div |N|$, 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 $deg(CTG)$ 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 $C_{separated}$ denotes the number of isolated components that do not connect to any other; $C_{mainNot}$ denotes the number of components that are not reachable from the default entry, usually the `MainActivity`; and $C_{exportNot}$ 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 $IC3_{Dial}$ and *StoryD* extend *IC3*, i.e., are IC3-based tools, they do not modify the inner models. As $IC3_{Dial}$ 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 $IC3_{Dial}$ 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.

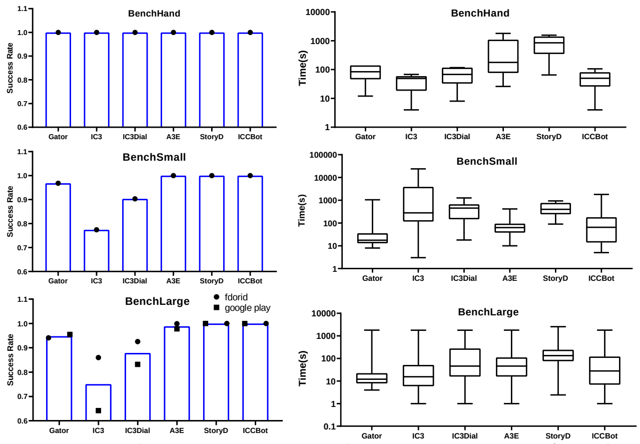

Figure 3: The Success Rate and Execution Time of Tools

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., $IC3_{Dial}$ 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 $deg(CTG)$, $C_{separated}$, $C_{mainNot}$ and $C_{exportNot}$ 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

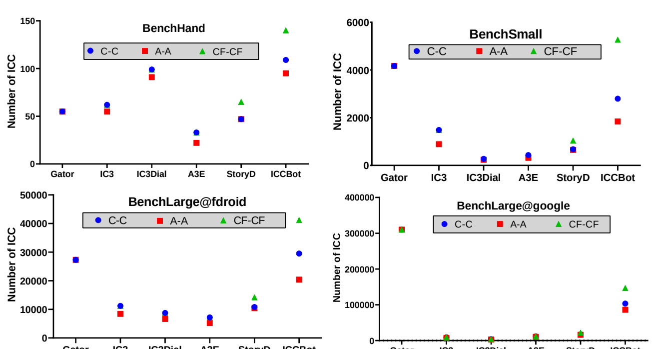

Figure 4: Evaluating with Number-based Metrics

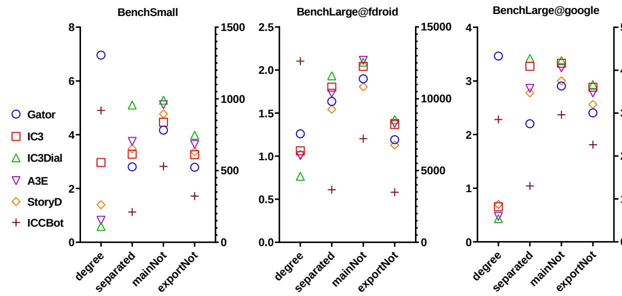

Figure 5: Evaluating with Graph-based Metrics

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 $IC3_{Dial}$ generates many FPs while *IC3* and *Gator* also generate a few ones. 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 calls 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 *BenR* as it contains various atypical types of ICC usage, which requires the modeling of specific APIs. *BenI* also leads to many FNs because it uses several not commonly used callback methods. Overall, on *BenchHand*, *StoryD* and A^3E behave well on FP rate, *ICCBot* behaves well on both FP and FN rates.

Table 5: Evaluating with Oracle on *BenchHand*

Bench	#OR	#FP / #FN					
		Gator	*IC3*	$IC3_{Dial}$	A^3E	*StoryD*	*ICCBot*
BenD	12	1/5	0/4	0/4	0/10	0/9	0/2
BenI	26	0/22	0/16	0/3	0/19	0/19	0/0
BenS	37	0/4	0/4	0/4	0/37	0/1	0/0
BenR	24	0/24	0/23	0/23	1/1	0/24	1/0
BenT	11	3/4	3/4	24/1	0/11	0/10	0/0
Sum	**110**	**4/59**	**3/51**	**24/35**	**1/78**	**0/63**	**1/2**

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 the 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 $IC3_{Dial}$ 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

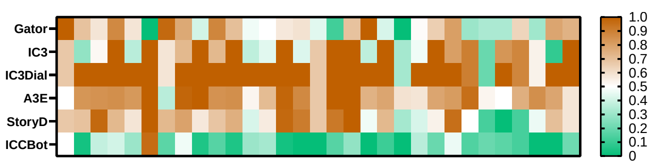

Figure 6: False Negative Rates of Tools on *HandSmall*

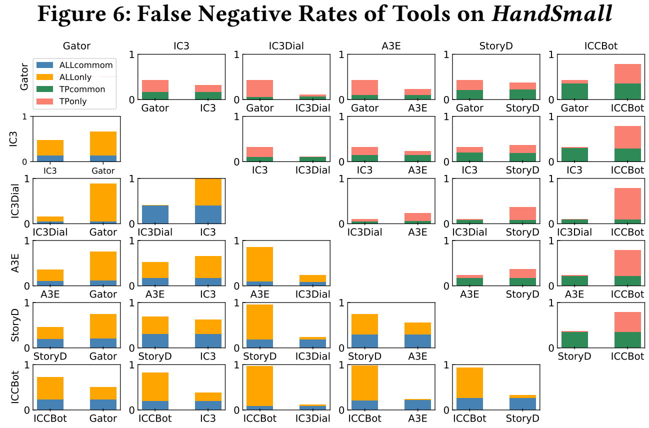

Figure 7: Pairwise Comparison about Reported and TP ICCs

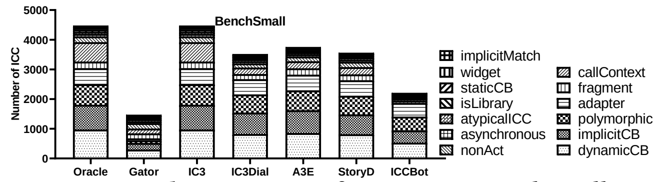

Figure 8: Characteristics of FN ICCs on *BenchSmall*

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*, $IC3_{Dial}$ 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

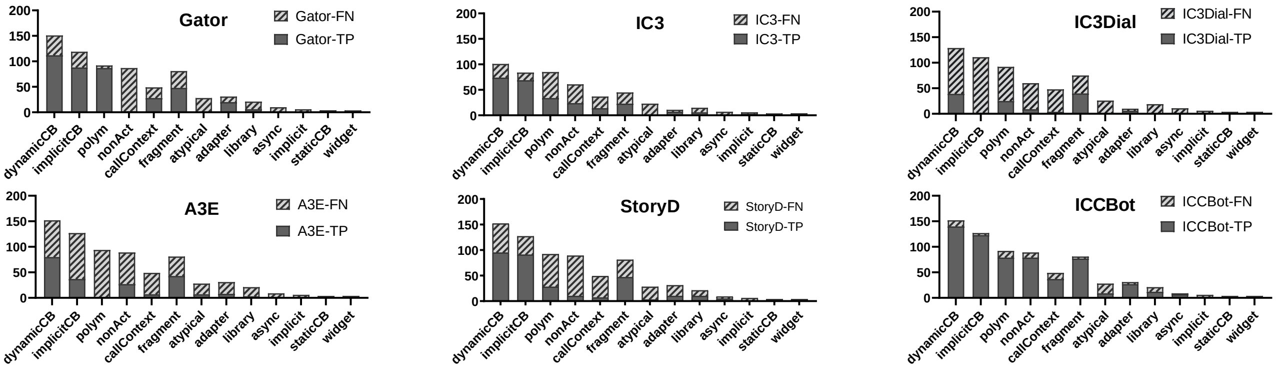

Figure 9: Number of FN and TP ICCs related to Single Characteristic on *BenchSmall*

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 researchs 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 $IC3_{Dial}$ can be adopted as they concern fragment and entry-point identification, respectively. Note that $IC3_{Dial}$ 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

Requirement & Candidate Tools	Requirement & Candidate Tools
Less time cost → *Gator, A^3E, ICCBot*	IC3-based update → *StoryD, IC3Dail, IC3*
More real ICCs → *ICCBot, Gator, StoryD*	UI analysis → *StoryD, Gator*
Fewer fake ICCs → *ICCBot, StoryD, IC3*	Intent field extract → *ICCBot, IC3, Gator*
High SuccRate → *ICCBot, StoryD, A^3E*	Fragment-aware → *ICCBot, StoryD*

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

```
//In the layout file of Component A.class (A to Tgt)
<Button android:id="@+id/button" android:onClick="onMyClick" />
<com.pkg.navView android:id="@+id/navView" />
//In com.pkg.navView.class
setNavigationItemSelectedListener(new OnNavigationItemSelectedListener(){
    public void onNavigationItemSelected(View v){
        startActivity(new Intent(com.pkg.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
    <intent android:targetPackage="com.pkg" android:targetClass="com.pkg.Tgt"/>
</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 call graph construction as well as a multiple-turn callback analysis.

Listing 2: FN: Multi-step Callback

```
// In Component A.class (A to Tgt)
pendingRunnable = new Runnable() {
    public void run() {
        addListener(new DrawerToggle(){
            public void onDrawerOpened(View v){
                startActivity(new Intent(Tgt.class));}});
button.setOnClickListener(new OnClickListener(){
    public void onClick(View v){
        new Handler().post(pendingRunnable); }});
```

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

```
// 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 can not 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

```
// In Component A.class (A to Tgt)
public void onCreate(){
    mViewPager.setAdapter(mAdapter);
    mAdapter = new FragmentPagerAdapter( getSupportFragmentManager()) {
        public Fragment getItem(int position) {
            switch (position){ case 0: return new F(); ...}}};}
// In Fragment F.class
public void onAttach(Activity act) {
    startActivity(new Intent(Tgt.class));}
```

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 $A \rightarrow Tgt$. 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

```
// 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(); }
```

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* (IC3, 22.5), etc. For these apps, we carefully read their code and infer why a nonexistent 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, A^3E 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

```
// In abstract class Father.class
public void onCreate() { super.onCreate(); fatherMethod(); }
abstract public void fatherMethod();
// In Activity SonA.class (SonA extends Father)
public void onCreate() { super.onCreate(); }
public void fatherMethod() {
    startActivity(new Intent(this, SonB.class)); }
// In Activity SonB.class (SonB extends Father)
public void onCreate() { super.onCreate();
    Intent received = getIntent();
    startActivity(new Intent(this, Tgt.class)); }
public void fatherMethod(){ /** do nothing **/}
// In Activity SonC.class (SonC extends Father)
public void onCreate() { super.onCreate(); }
public void fatherMethod(){ /** do nothing **/ }
```

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 ($A \rightarrow D$, $B \rightarrow C$) are FPs.

Listing 7: FP: Decorator Method

```
// In Class Util.class
public static void launchAct(Context ctx, Intent i) {
    addFlagForIntent(i); ctx.startActivity(i);}
// In Component A.class
public void onCreate(){ Util.launchAct(new Intent(getBaseContext(), C.class));}
// In Component B.class
public void onCreate(){ Util.launchAct(new Intent(getBaseContext(), D.class));}
```

P8: Type Inference. In Listing 8, component A dynamically rigisters two broadcast receivers and set 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

```
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")); ...}}
```

5.3 Handling of FN/FP Patterns

Among the five FN patterns, both patterns P1 and P2 are callback-related. Meanwhile, the identification of a single callback also leads to many FNs. Tool developers could extend their callback identification module to handle these specific cases, for which the key challenge is how to automatically identify the layout-related and user-customized callbacks precisely. Pattern P3 depends on whether the analysis approach is path- and context-sensitive. The handling of this pattern is related to the design of the tool and may need more effort. Besides these patterns, the atypical ICC leads to FNs on many tools. Developers could quickly update the exit method set to support this characteristic. It is also not hard to extend tools to support non-Activity components, e.g., Service. However, performing extension around fragment, container (P4), and inter-procedural assignment (P5) is not easy and requires fine-grained models. For the FP patterns, developers could adopt more precise call graph and type inference analysis algorithm to avoid P6 and P8, while to reduce the FPs related to P7, context-sensitive analysis is required.

6 THREATS TO VALIDITY

In this section, we discuss the threats to the validity faced by this work. The threats to external validity relate to the generalizability of the experimental results. Our oracles for real-world apps are extracted from 31 benign Android projects on the public markets, while the results may not generalize beyond the 31 apps, especially the malicious apps. Threats to internal validity concern factors internal to our approach. We manually confirm the correctness of the dynamically reported ICC links and label the related characteristics, which might introduce bias. To mitigate this risk, 18 tag inference checkers are designed for double-checking. For ICCs that can be triggered by multiple paths on the sliced code, we only record and label one path, which may influence the evaluation based on these labels. Although this type of bias is difficult to avoid, we try to cover more ICCs to reduce the accidental errors brought by it.

7 CONCLUSION

Identifying ICC links precisely is essential to the analysis of apps. However, the comprehensive evaluation of Android ICC resolution techniques faces big challenges due to the lack of high-quality datasets and metrics. In this paper, we present multiple-type benchmark suites and design corresponding evaluation metrics. For the oracle construction on real-world apps, we propose a dynamic ICC extraction approach and combine an automatic result filter and careful manual code auditing. With both the constructed oracle set and the proposed metrics, we identified 38%-85% ICCs that are missed by tools and observed many wrongly reported ICCs. Finally, based on the labeled characteristic tags, we discover the pros and cons of the state-of-the-art tools and summarize eight common FN/FP patterns for further improvement.

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 and Grant No. 62132020), the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036), the Guangdong Basic and Applied Basic Research Fund (Grant No. 2021A1515011562), and the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001).

REFERENCES

[1] A3E. 2016. A3E. https://github.com/tanzirul/a3e.

[2] Waqar Ahmad, Christian Kästner, Joshua Sunshine, and Jonathan Aldrich. 2016. Inter-app communication in Android: developer challenges. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, Miryung Kim, Romain Robbes, and Christian Bird (Eds.). ACM, 177–188.

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, 468–471.

[4] 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.

[5] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA 2013, part of SPLASH 2013*. 641–660.

[6] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *TSE* 41, 9 (2015), 866–886.

[7] Shweta Bhandari, Frédéric Herbreteau, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, and Partha S. Roop. 2020. *SneakLeak+*: Large-scale klepto apps analysis. *Future Gener. Comput. Syst.* 109 (2020), 593–603.

[8] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. 2017. Android inter-app communication threats and detection techniques. *Comput. Secur.* 70 (2017), 392–421.

[9] Zohreh Bohluli and Hamid Reza Shahriari. 2018. Detecting Privacy Leaks in Android Apps using Inter-Component Information Flow Control Analysis. In *15th International ISC (Iranian Society of Cryptology) Conference on Information Security and Cryptology, ISCISC 2018, Tehran, Iran, August 28-29, 2018*. IEEE, 1–6.

[10] 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 *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*. ACM, 71–85.

[11] Huan Chang, Lingguang Lei, Kun Sun, Yuewu Wang, Jiwu Jing, Yi He, and Pingjian Wang. 2021. Vulnerable Service Invocation and Countermeasures. *IEEE Trans. Dependable Secur. Comput.* 18, 4 (2021), 1733–1750.

[12] Sen Chen, Lingling Fan, Chunyang Chen, and Yang Liu. 2022. Automatically Distilling Storyboard with Rich Features for Android Apps. In *IEEE Transactions on Software Engineering (TSE)*. IEEE.

[13] 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.

[14] Chris Chao-Chun Cheng, Chen Shi, Neil Zhenqiang Gong, and Yong Guan. 2021. LogExtractor: Extracting digital evidence from android log messages via string and taint analysis. *Digit. Investig.* 37 Supplement (2021), 301193.

[15] Component. 2022. Component. https://developer.android.com/guide/components/fundamentals#Components.

[16] DroidBench. 2017. DroidBench. https://github.com/secure-software-engineering/DroidBench.

[17] Karim O. Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara G. Ryder. 2020. Identifying Mobile Inter-App Communication Risks. *IEEE Trans. Mob. Comput.* 19, 1 (2020), 90–102.

[18] F-Droid. 2019. https://f-droid.org/.

[19] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 486–497.

[20] Fragment. 2022. Fragment. https://developer.android.com/guide/fragments.

[21] GATOR. 2019. GATOR. http://web.cse.ohio-state.edu/presto/software/gator/.

[22] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. 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.

[23] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 269–280.

[24] IC3. 2015. IC3. https://github.com/siis/ic3.

[25] IC3-DIALDroid. 2020. IC3-DIALDroid. https://github.com/dialdroid-android/ic3-dialdroid.

[26] ICC-Bench. 2017. ICC-Bench. https://github.com/fgwei/ICC-Bench.

[27] ICC-Resolution-Evaluation. 2022. ICC-Resolution-Evaluation. https://github.com/hanada31/ICC-Resolution-Evaluation/.

[28] ICCBot. 2022. ICCBot. https://github.com/hanada31/ICCBot.

[29] ICCViewer. 2022. ICCViewer. https://iccviewer.ldby.site/ICCViewer/.

[30] Intent. 2022. Intent. https://developer.android.com/guide/components/intents-filters.

[31] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. 2019. Search-based energy testing of Android. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 1119–1130.

[32] Ajay Kumar Jha and Woo Jin Lee. [n.d.]. ICCMATT: Modeling, analysis, and test case generation of inter-component communication in Android. ([n. d.]).

[33] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. 2012. SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. *MoST* 12, 110 (2012), 1.

[34] Duling Lai and Julia Rubin. 2019. Goal-Driven Exploration for Android Applications. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 115–127.

[35] Youn Kyu Lee, Jae Young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A *SEALANT* for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 312–323.

[36] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE 2015*. 280–291.

[37] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.* 88 (2017), 67–95.

[38] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM, 229–240.

[39] 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.

[40] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick D. McDaniel. 2016. Composite Constant Propagation and its Application to Android Program Analysis. *IEEE Trans. Software Eng.* 42, 11 (2016), 999–1014.

[41] 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 *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. USENIX Association, 543–558.

[42] Google Play. 2019. https://play.google.com/store.

[43] RAICC-Bench. 2021. RAICC-Bench. https://github.com/Trustworthy-Software/RAICC.

[44] Alireza Sadeghi, Reyhaneh Jabbarvand, Negar Ghorbani, Hamid Bagheri, and Sam Malek. [n.d.]. A temporal permission analysis and enforcement framework for Android. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). 846–857.

[45] Jordan Samhi, Alexandre Bartel, Tegawendé F. Bissyandé, and Jacques Klein. 2021. RAICC: Revealing Atypical Inter-Component Communication in Android Apps. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1398–1409.

[46] StoryDistiller. 2022. StoryDistiller. https://github.com/tjusenchen/StoryDistiller.

[47] StoryDroid-Bench. 2019. StoryDroid-Bench. https://sites.google.com/view/storydroid/.

[48] Yutian Tang, Yulei Sui, Haoyu Wang, Xiapu Luo, Hao Zhou, and Zhou Xu. [n.d.]. All your app links are belong to us: understanding the threats of instant apps based attacks. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). 914–926.

[49] Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-an, Gregg Rothermel, and Jackson Dinh. 2017. An efficient, robust, and scalable approach for analyzing interacting android apps. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 324–334.

[50] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on*

Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014. ACM, 1329–1341.

[51] Jiwei Yan, Hao Liu, Linjie Pan, Jun Yan, Jian Zhang, and Bin Liang. 2020. Multiple-entry testing of Android applications by constructing activity launching contexts. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. 457–468.

[52] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. ICCBot: Fragment-Aware and Context-Sensitive ICC Resolution for Android Applications. In *The 44th International Conference on Software Engineering, ICSE 2022 (Tool Track)*.

[53] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 89–99.

[54] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *ASE 2015*. 658–668.

[55] Zhemin Yang and Min Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*. 101–104. https://doi.org/10.1109/WCSE.2012.26

[56] Jie Zhang, Cong Tian, and Zhenhua Duan. 2021. An efficient approach for taint analysis of android applications. *Comput. Secur.* 104 (2021), 102161. https://doi.org/10.1016/j.cose.2020.102161

[57] Jie Zhang, Cong Tian, Zhenhua Duan, and Liang Zhao. 2021. RTPDroid: Detecting Implicitly Malicious Behaviors Under Runtime Permission Model. *IEEE Trans. Reliab.* 70, 3 (2021), 1295–1308.

[58] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society.