ICC-Inspect: Supporting Runtime Inspection of Android Inter-Component Communications

John Jenkins
Washington State University, Pullman, WA
john.jenkins@wsu.edu

Haipeng Cai
Washington State University, Pullman, WA
hcai@eecs.wsu.edu

ABSTRACT

We present ICC-Inspect, a tool for understanding Android app behaviors exhibited at runtime via inter-component communication (ICC). Through lightweight Intent profiling, ICC-Inspect streams run-time ICC information to a dynamic visualization framework which depicts interactive ICC call graphs along with informative ICC statistics. This framework allows users to examine the details of a specific fragment of execution in the context of the holistic ICC view of an app. Through its ability to concisely map in a visual format the complex ICC mechanisms of any Android app, ICC-Inspect facilitates behavior understanding and debugging of Android programs. The open-source release, documentation, and a video demo of ICC-Inspect are available here.

KEYWORDS

Android, ICC, behavior, understanding, inspection

ACM Reference Format:
John Jenkins and Haipeng Cai. 2018. ICC-Inspect: Supporting Runtime Inspection of Android Inter-Component Communications. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3197231.3197233

1 INTRODUCTION

Android is now the leading mobile computing platform having secured nearly 85% mobile market share [8]. As a result, more and more software applications today are Android apps. In this context, it is increasingly crucial to support app developers with *various* automated tools for producing apps with higher quality and lower cost. Also, as securing Android apps against malicious behaviors has become a growing concern [2, 10], tools that help developers understand how apps are coded and executed are essential for longer-term and proactive security defense.

Android application programming features the use of a communication model, called inter-component communication (ICC), by which components in an app (resulting from functional modularization) collaborate toward achieving higher-level tasks of the app. An Android app typically uses messaging objects called *Intents* as the primarily means to perform ICC, interacting with components internal or external to the app to accomplish any task a calling component is unable to complete by itself. Thus, resolving ICC Intents and mapping ICCs among components has been a requirement of performing static or dynamic analysis of Android apps [10, 11]. Also, a key step in understanding the behaviors of any Android app involves understanding ICCs within the app. Yet, understanding ICCs is currently a major challenge to mobile developers, especially within the context of undocumented third-party APIs [1]. Further, there is a tendency for malware to take advantage of the ICC flexibility in order to bypass security mechanisms in Android (e.g., through Intent spoofing and hijacking) [4, 11]. Inspecting the ICC-induced behaviors against such threats can help mobile developers produce securer apps.

While there has been considerable progress made in coherent ICC analysis frameworks, to date most existing approaches report conservative (hence large) ICC analysis results in a textual format [6, 7]. Not only are the textual results difficult to navigate, they are mostly *static* thus provide few information on how the ICCs are actually invoked in the app and, more importantly, how the app *behaves* with respect to the ICCs that are *exercised at runtime*.

As a means of contextualizing the massive amount of information that describes an application's ICC, we developed ICC-Inspect, a tool that assists with Android app behavior understanding.
ICC-Inspect specifically focuses on facilitating the inspection of ICCs through a synergy of static and dynamic ICC analysis along with interactive, dynamic information visualizations. By incorporating the bytecode analysis utilities of Soot [5], the static analysis module of ICC-Inspect inserts in a given app probes linked to run-time monitors that capture and transmit the information of exercised ICCs as the Android application is traversed by the app user. These transmissions are then received and processed by the visualization framework of ICC-Inspect, where a dynamic call graph is built from the ICC traces. The call graph is then automatically updated as the user continues to traverse the instrumented app.

An optional static call graph is also generated during the static analysis, which contains every potential ICC in the target app. The static ICC calling relationships are established conservatively based on the app's Intent filters as specified in its manifest file. If the user is interested in exploring an individual Intent's fields, detailed ICC information can be displayed by means of the user hovering over individual ICC Intent nodes. All Intent resolution that occurs throughout the app's traversal is cleanly depicted using a variety of colored edges, which indicate the specific type of Intent resolution that has occurred. Edge labels further detail the specific matching Intent fields that lead to the Intent resolution. Alongside the ICC graphs, detailed tables break down specific ICC statistics. In addition, the static graph is interactively linked to the dynamic graph: if the user would like to explore the relationship between a

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5712-8/18/05.
https://doi.org/10.1145/3197231.3197233

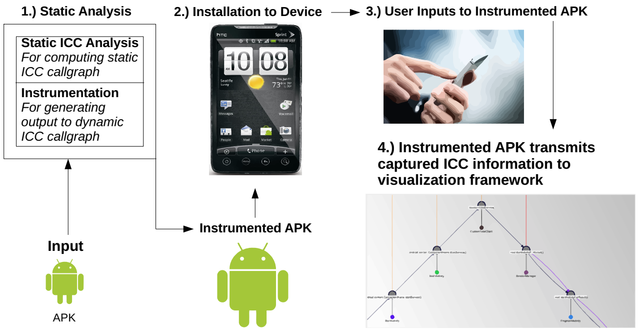

1.) Static Analysis
2.) Installation to Device
3.) User Inputs to Instrumented APK
Static ICC Analysis
For computing static ICC callgraph
Instrumentation
For generating output to dynamic ICC callgraph
4.) Instrumented APK transmits captured ICC information to visualization framework
Instrumented APK
Input
APK

Figure 1: Overview of ICC-Inspect process flow.

dynamic ICC and its static counterpart, all they need to do is click the specific dynamic ICC node, and the node in the static ICC graph that corresponds to that particular node will be highlighted.

2 THE ICC-INSPECT APPROACH

We first give an overview of ICC-Inspect and then elaborate on the key modules of ICC-Inspect: static analysis, profiling and dynamic analysis, and interactive visualization.

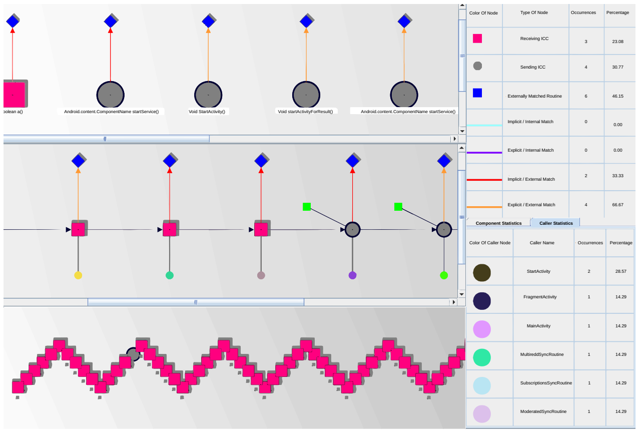
2.1 ICC-Inspect Architecture

The architecture of ICC-Inspect is depicted in Figure 1. The input to ICC-Inspect is a target Android app for inspection, and the output is the interactive dynamic visualization of ICCs being exercised by user inputs to the app. The intermediate processing consists of three major phases in the four numbered steps. The static ICC analysis (step 1) takes the APK of the input app and instruments the APK for ICC profiling. Static ICC analysis is also performed during this phase to build the static ICC call graph. The instrumented APK is then fed into the profiling phase (steps 2 and 3), where traces are collected during app executions driven by the user inputs to the app, and then used to construct the dynamic call graph. Finally, the interactive visualization phase takes the ICC call traces to construct the dynamic call graphs, and visualize both the static and dynamic views of the ICCs. The user can henceforth interact with the app and the visualizations.

2.2 Static Analysis

Static ICC analysis. Our static analysis starts with building the static ICC call graph of the given app. The static graph is considered optional in regards to ICC-Inspect static analysis. Depending on the type of analysis the user wants to perform with ICC-Inspect, the static graph may be unnecessary, and hence omitting the extra processing time required by the static graph construction will reduce the overall time cost of the entire ICC-Inspect analysis. If the user elects to not have ICC-Inspect depict a static ICC graph, the dynamic graph will then occupy the additional space that the static graph would consume.

As depicted in Figure 2 (bottom left), an interactive static call graph of the app under analysis is provided below two dynamic ICC graphs. The motivation is to provide the user a reference of every potential ICC within the scope of the app. The static graph can also be used in tandem with the dynamic graph, for example; clicking on an ICC node in either dynamic graphs will highlight and center the corresponding ICC node in the static graph. As the static call graph will often be depicting a plethora of ICC nodes, an alternating 45-degree zigzag pattern is designed to maximize use of available space. IC3 [6] is used to resolve ICC Intent fields at each ICC call site in the app. For each of the ICCs whose target component is resolved as a valid component name, the ICC is *explicit* and the static ICC call edge is established (from the enclosing component of the ICC call to the target component). All other ICCs are *implicit*, for which the manifest file of the app is parsed to extract the Intent filters of each component declared. By matching the Intent filters against the resolved Intent fields at all ICC call sites, potential calling relationships between two components are identified and the static ICC call edges established accordingly. This matching is conservative in order to avoid missing any possible ICC call edges. In contrast, a dynamic call graph edge is established when an Intent at an ICC receiving site exactly matches an earlier Intent from an ICC invocation site in the ICC trace. This matching is precise because all Intent fields are exactly resolved at runtime.


oolean a()
Android.content.ComponentName startService()
Void StartActivity()
Void startActivityForResult()
Android.content.ComponentName startService()
Component Statistics
Caller Statistics

Color Of Node	Type Of Node	Occurrences	Percentage
🟥	Receiving ICC	3	23.08
●	Sending ICC	4	30.77
🟦	Externally Matched Routine	6	46.15
—	Implicit / Internal Match	0	0.00
—	Explicit / Internal Match	0	0.00
—	Implicit / External Match	2	33.33
—	Explicit / External Match	4	66.67

Color Of Caller Node	Caller Name	Occurrences	Percentage
●	StartActivity	2	28.57
●	FragmentActivity	1	14.29
●	MainActivity	1	14.29
●	MultireddSyncRoutine	1	14.29
●	SubscriptionsSyncRoutine	1	14.29
●	ModeratedSyncRoutine	1	14.29

Figure 2: The four visual components of ICC-Inspect.

Instrumentation. To generate a dynamic call graph of ICC that updates as the user traverses the app, ICC-Inspect continuously collects ICC call traces and *streams* them to our visualization framework. To that end, a probe for monitoring Intent objects is inserted after each ICC invocation and receiving site found in the app's bytecode. The probe is hooked to a run-time monitor that reports the run-time value of each Intent field and transmits the values to the visualization framework through TCP socket-based messaging.

2.3 Profiling

Setup. After the static-analysis phase, the instrumented APK is installed onto either a handheld device or an emulator [3]. We implemented inter-process messaging based on a TCP socket as the primary mechanism for trace transmission to allow for using ICC-Inspect on a physical Android device.

Tracing. As the user triggers specific events in the target app that cause ICCs, the relevant Intent information will be generated by the run-time monitor hooked by the probes inserted in the app. Then, the traces are sent to the visualization framework via the socket-based messaging mentioned earlier.

2.4 Interactive Dynamic Visualization

Once the static ICC call graph is built (if opted for) and the (instrumented) app is installed, the visualization will start. Upon startup,

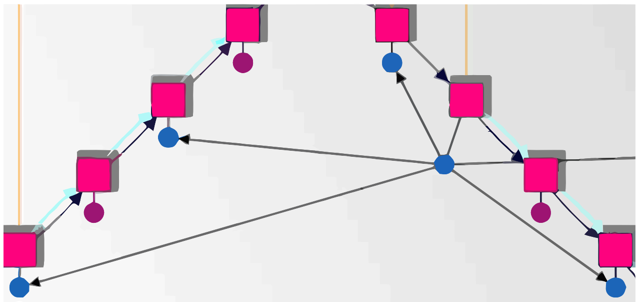


Figure 3: Examining the calling pattern of Intent callers by activating a specific caller node's matching edges.

ICC-Inspect's main window will be rendered to the screen with two empty dynamic ICC call graphs, a static ICC graph (if selected by the user), and two legends to the right of the graphs (see Figure 2). The dynamic graph in the middle of the window handles overall dynamic graph structure, while the upper graph focuses context from the central graph into exploring the individual ICC pairs themselves. The legend in the top-right corner serves as a reference for the ICC nodes, and resolution edges in the dynamic ICC graph. Specifically, each node and edge type is depicted by four properties: coloration, specific name, number of occurrences, and statistical percentage of occurrence within the evolution of the dynamic graph. The legend in the bottom-right corner serves as a reference to all of the caller nodes within the graph. As in the ICC legend, individual callers are depicted by the same four properties.

When an individual Intent is received for processing from the instrumented app, an Intent listener first parses all the individual fields of the Intent into a custom data structure (referred to as *Intent-Type*), for ease of processing during dynamic call graph generation. The IntentType is then passed into the main graph construction, and an ICC node is generated from the IntentType, with coloration and shape matching the type of ICC node that is being processed (sending or receiving). The ICC node's label is instantiated to display the Intent's callsite information. A unique numeric ID will also be generated and overlaid on each ICC link.

As an individual ICC node's specific field information is often too dense to display all at once, additional fields can be viewed as needed by the user. For instance, the graph constructor uses a hashmap to link the ICC node's internal ID with the specific IntentType the ICC node is based off of. If the user wants to access analysis results for a specific ICC, they can then hover their mouse over the distinct ICC node. The mouse hover triggers the event handler to retrieve the specific IntentType that generated the ICC node via the hashmap, and then displays every individual field to a translucent window positioned next to the ICC node, that will subsequently disappear whenever the user moves the mouse off of the selected ICC node.

When an ICC node is created, a smaller caller node is also created directly below the ICC node, and is linked to the ICC node via an edge. The caller node represents the calling method of the specific Intent that was mapped into an ICC node. If the user clicks on the caller node, edges are automatically generated that lead to every other caller node in the dynamic graph of the same type. The intention of the caller node generating edges is to provide the user with a high-level overview of a caller's calling pattern throughout the dynamic graph's evolution, as illustrated in Figure 3.

As soon as the dynamic graph includes at least two ICC nodes, time-stamp edges and Intent resolution edges are automatically generated and linked respectively between individual ICC nodes. The time-stamp edge lists as its label the time difference between the current ICC node being generated, and it's predecessor. If multiple ICC links are generated at very close intervals, they will still be identifiably different by their uniquely generated numeric ID's in the graph. The resolution edge compares individual Intent fields between the ICC node's predecessor and itself to determine if an intra-app ICC match has occurred. If no intra-app ICC match is detected, the current ICC node will instead be linked to an external routine, which will cause the generation of an external ICC node. The resolution details can be accessed by the user hovering their mouse over the specific external ICC node.

3 USAGE OF ICC-INSPECT

To illustrate the usage of our tool, we show how ICC-Inspect is used to investigate obfuscated ICCs in an app. In particular, we explore Sonic Dash, an app where the majority of ICCs are dedicated exclusively to advertisement services. After starting the app, the dynamic trace will contain several ICCs with obfuscated call-sites. The user may first investigate the caller of these ICCs by clicking on the caller node below the obfuscated ICC, which will then produce edges to every other ICC that the caller has also instantiated (Figure 4). If the user would like to further investigate a specific ICC, they can simply hover mouse over the relevant node, and a translucent window will pop up containing all of the ICC's

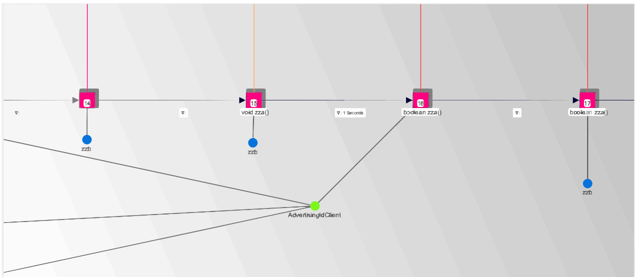
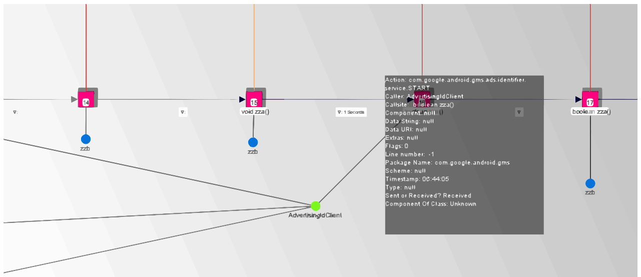


Figure 4: ICC-Inspect at first glance. It is apparent that the receiving ICC call-site has been obfuscated.



```
Action: com.google.android.gms.ads.identifier.
service.START
Caller: AdvertisingIdClient
Callsite: boolean zza()
Component: null
Data String: null
Data URI: null
Extras: null
Flags: 0
Line number: -1
Package Name: com.google.android.gms
Scheme: null
Timestamp: 00:44:05
Type: null
Sent or Received? Received
Component Of Class: Unknown
```

Figure 5: By hovering the mouse over the ICC being investigated, a window pops up detailing specific information that was mined along with the ICC in the instrumented APK. We can see from the "Caller" field that this ICC is related to an advertisement service: "AdvertisingIdClient".

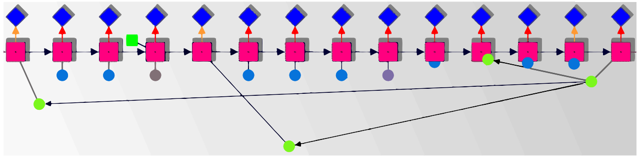


Figure 6: By zooming out, and clicking on the ICC's caller node, we can begin to explore the other ICC's being set by the same caller. This will allow us to begin exploring the overall relationship that this specific caller has with the dynamic ICC trace. Note that all nodes in ICC-Inspect can be dropped and dragged to any point in the graph. The motivation behind this interaction is to allow the user to re-configure the trace to fit whatever abstraction they are trying to achieve for their application's trace.

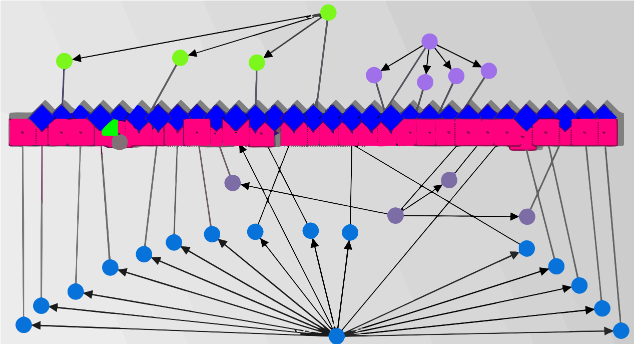


Figure 7: By zooming out further and clicking on all other available callers, we get an exploded view of every callers relationship with the dynamic ICC trace of the application.

detailed information (Figure 5). It is apparent that the ICC is related to advertising services in this instance, as its detailed information window specifies a caller field of 'AdvertisingIDClient'. If the user would like to get a quick visual representation of the frequency of the AdvertisingIdClient's ICCs so far in the trace, they can quickly zoom out, and will be able to observe specifically where in the trace these ICCs occur by means of the caller node's edges (Figure 6). Finally, if a user would like a visual depiction of the frequency of all callers in the dynamic trace, they can simply click on the respective caller nodes and re-arrange them into whichever configuration most concisely helps them understand the overall caller frequency in the graph (Figure 7).

Tool performance. We applied ICC-Inspect to various popular apps downloaded from Google Play. The standard test case for each app consisted of manually navigating the app through to completion, while observing results produced in the static and dynamic ICC graphs. The main cost lies in the time for static analysis, especially for constructing the static ICC graph, if it is opted for. For a few dozens of apps, the static analysis took on average no more than 3 and 15 minutes without and with the static ICC graph, respectively. Note that for a given app, the static analysis cost is incurred only once. After this phase, the user can explore the app with ICC-Inspect without rerunning the static analysis. Other costs (for tracing and rendering) of our tool were negligible. The total run time of the tool depends on how long the user wants to manipulate the app with ICC-Inspect.

4 RELATED WORK

Epicc [7] employs interprocedural dataflow analysis to statically resolve field values of ICC Intents hence match ICC across apps to identify ICC-induced security vulnerabilities. Later, IC3 [6] achieved higher precision in Intent resolution than Epicc through multi-valued constant propagation. These tools address ICC resolution to empower ICC-involved client analyses, such as malware detection and taint analysis [4, 11]. We used IC3 for our static ICC analysis. There are also numerous tools built on ICC analysis for identifying vulnerable ICCs, as surveyed in [10]. In contrast, our tool primarily aims at understanding ICCs themselves as part of program comprehension of Android apps. Although in one of our case studies we used this understanding to help capture malicious behaviors, our tool supports better Android program comprehension in general. The tool based on COVERT [9] visualizes ICCs, but to help users better understand how the ICC-related security vulnerabilities come about. Also, the visualization is static and based on static ICC analysis results, as opposed to our focus on dynamic visualization showing how ICCs are *exercised* in a given app.

5 CONCLUSION

We presented ICC-Inspect, a tool for facilitating examination of Android app behaviors with respect to exercised ICCs via interactive, dynamic ICC call graphs. Since ICC has been an obstacle for Android app understanding and a main surface of app security threats, ICC-Inspect potentially empowers both behavior understanding and security inspection of Android apps by mobile developers. We illustrated how our tool can be used for such purposes and revealed the practical usage overheads expected.

REFERENCES

[1] Waqar Ahmad, Christian Kästner, Joshua Sunshine, and Jonathan Aldrich. 2016. Inter-app communication in Android: Developer challenges. In *MSR*. 177–188.

[2] Haipeng Cai and Barbara G Ryder. 2017. Understanding Android application programming and security: A dynamic study. In *ICSME*. 364–375.

[3] Google. 2017. Android emulator. http://developer.android.com/tools/help/emulator.html. (2017).

[4] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *International Workshop on the State of the Art in Java Program Analysis (SOAP)*. 1–6.

[5] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot - a Java Bytecode Optimization Framework. In *Cetus Users and Compiler Infrastructure Workshop*. 1–11.

[6] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE*. 77–88.

[7] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *USENIX Security Symposium*. 543–558.

[8] International Data Corporation (IDC) Research. 2018. Android dominating mobile market. http://www.idc.com/promo/smartphone-market-share/. (2018).

[9] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2015. Analysis of Android inter-app security vulnerabilities using COVERT. In *ICSE*, Vol. 2. 725–728.

[10] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The Evolution of Android Malware and Android Analysis Techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 76.

[11] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*. 1329–1341.