Dissecting Android Inter-Component Communications via Interactive Visual Explorations

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

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

***Abstract*—Inter-component communication (ICC) serves as a key element of any Android app's implementation. Specifically, an Android app uses Intents as the main mechanism for ICC to complete tasks such as switching between different user interfaces, starting background services, communicating to other apps on the Android device, and saving or retrieving data from device storage. Thus, dissecting how an app uses ICCs to accomplish its tasks is fundamental to understanding the app's underlying behaviors. Existing works involving ICCs focus on resolving Intents and/or mapping ICCs for security analysis purposes. While the ICC analysis result is potentially informative, it is difficult to digest on its own and has not been utilized for app/ICC comprehension. Also, the result is based on static analysis, and thus does not inform of run-time app behaviors exercised via ICCs. We propose the first approach to dissecting Android ICCs via interactive, dynamic visualizations, empowered by static and dynamic ICC analysis combined. Through multiple semantically linked views and in-situ interaction features, our approach enables real-time visual explorations of ICCs as they are triggered by user inputs to the app under analysis. It conveys rich ICC information while managing limited visual space through various visualization design strategies. Our case studies with a number of commonly used apps have showed promising merits of the approach for both deep ICC comprehension and security vulnerability inspection, as well as practical scalability. Our tool prototype of the approach has enabled quick revelation of inefficient, intrusive, and malicious behaviors in several popular apps that were normally hidden to users.**

I. Introduction

As the Android ecosystem increasingly dominates the mobile computing market [1], it is crucial for developers to understand effective development and security defense strategies for Android applications (*apps*). A prominent characteristic of Android apps lies in their common uses of a communication model called inter-component communication (ICC). While ICC has largely contributed to app development productivity through code reuse and implementation flexibility, it also has been a main surface for security attacks in Android [2]–[6]. ICCs can expose apps to security threats mainly because the flexibility they offer also allows malicious apps to either exploit benign ones [7] or collude with other malicious apps [8].

ICCs are a major obstacle for app understanding as well [7], [9], [10]. One reason is that ICCs are of various types, based on the communication scope and linkage specificity [9]. In addition, ICCs can possibly carry complicated payloads, and link components both within individual apps and across different apps [7]. Moreover, ICCs involve sophisticated intermediation of the Android platform [6]. The primary means for ICC is *Intent*, a messaging object that passes the communication content between components. Thus, dissecting Intent-based ICCs is an essential first step towards deeply and comprehensively understanding the behaviors of Android apps.

Various approaches have been proposed to resolve ICC Intents [10], [11] and map ICCs between components within or across apps [3], [12], [13]. These approaches are aimed at identifying security vulnerabilities in Android apps, using the ICC analysis underneath as an enabling technique. The resulting Intent data and ICC links among components have been further utilized for developing more security tools (e.g., learning-based malware detector [4] and versatile app classifiers [14]). However, they have not been employed for developing direct tool supports for more deeply understanding app behaviors. While potentially useful for high-level comprehension of ICC-related app features, the ICC information itself is not sufficient for deeply examining complex app behaviors exhibited through sophisticated intra- and inter-app ICCs.

Moreover, existing ICC analysis techniques are based on static code analysis, thus are subject to both imprecision and unsoundness due to common obstacles to static analysis (e.g., reflection and code obfuscation) [15]. Further, Intent fields cannot always be resolved statically [16] (e.g., the `extras` field). Since it is based on the resolved Intents, static ICC mapping is impeded alike. In addition, many ICCs have their target components determined by the Android platform at runtime. Static analysis has no way to precisely map such ICCs between components. These technical difficulties result in ICC information that is often too rough to be practically useful for understanding how ICC works in an app. Notably, the static ICC information does not provide an immediate means for understanding how Android apps *actually behave* with respect to the ICCs in the apps *exercised at runtime*.

In this paper, we propose a novel approach for deeply understanding Android app construction and behaviors with respect to ICCs. By combining lightweight ICC profiling and interactive dynamic visualizations, our approach aims at offering a practical and effective support for dissecting complex run-time interactions among app components. Given an Android app under analysis, we first instrument it by probing at all ICC API callsites in the app. At runtime, ICC traces are collected and fed into our visualization framework while the app is being manipulated. A dynamic ICC call graph is built on the fly based on the streaming ICC calls exercised by the user inputs, and is rendered in real time upon new ICC instances via dynamic graph visualization. In addition, by leveraging results of static Intent resolution and ICC mapping, the framework further provides a view of the static ICC graph of the app that is linked to the dynamic call graph view. The complete ICC mapping offers an important context for exploring and navigating in the dynamic visualization.

Through enabling simultaneous visual exploration and app navigation that are automatically synchronized, our approach

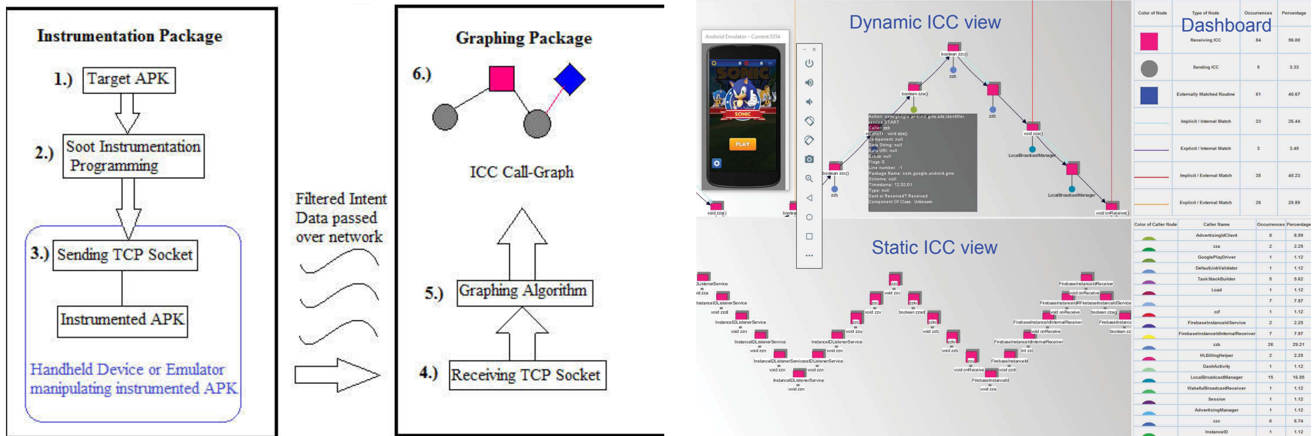

Fig. 1: Overview of our approach to dissecting ICCs: the technique workflow (left) and the user interface outlook (right).

empowers users to extensively inspect component-level interactions within and at the app boundary hence to deeply understand ICC-induced app behaviors. ICCs are commonly invoked both in app code and at runtime [16]. The large volume of ICCs, even only the exercised subset, can quickly constitute visual clutters in a graph-based visualization. The problem is further aggravated when all ICC call instances are to be visualized. To overcome these challenges, we propose various visual design strategies well-suited for ICC visualizations, combined with statistical ICC information summaries. The rich interaction features offered in our visualization framework, in addition to enhancing ICC understanding, further mitigate common visual-perception blockades by exploiting human decision making during the visualization exploration process.

To validate our design, we have developed a tool prototype for the proposed approach. We also have successfully applied this prototype to a set of top apps from Google Play in our preliminary evaluation. In several case studies, our approach allowed us to clearly observe and digest the app behaviors with respect to exercised ICCs. More importantly, it has enabled us to discover abnormally heavy workloads and resource usage of benign apps on advertisement services in one instance, and to reveal an inter-app security attack and understand how the attack is triggered and performed in another case.

In summary, the main contributions of this paper include:

- A novel approach to dissecting Android app behaviors with respect to Intent-based intra- and inter-app ICCs. To the best of our knowledge, this is the first attempt that focuses on addressing the unattended difficulties in understanding ICCs in Android apps.
- Visual design strategies and information summarization schemes that help mitigate visual exploration challenges in graph visualizations of sizable call traces.
- A prototype implementation of the proposed approach, and a preliminary evaluation for it that shows its practical scalability and usability. Using the prototype has revealed abnormal behaviors in widely used benign apps and demonstrated the detailed process of security attacks.

II. Background

The understanding of ICCs involves that of app components, Intents, and ICCs, of various types, and their relationships.

Component types. Android features a framework-based application development paradigm, by which an app may be quickly built by implementing four types of components each inherited from a top component (class) defined in the Android framework. Specifically, an *Activity* component represents a single screen of an app's user interface; a *Service* component represents an app running as a service in the background; a *Content Provider* component assists an app with data storage, data sharing, and data security; and a *Broadcast Receiver* component is that which is registered to be notified for specific application or system-wide events. Whether a type of component is included in an app depends on whether the app implements the corresponding scope of functionalities.

ICC and Intent types. For an Intent-based ICC, the associated Intent object carries the actual communication information between the two components involved. Intents can be thought of as the mechanisms of ICC that app components use to interact, both within an app and with other apps on the same device. An Android developer designs and sets an Intent when they need an operation to be performed by a component. There are two ICC types in terms of communication *scope*. An ICC is *internal* if the two components involved are defined in the same app, and *external* otherwise. Accordingly, the associated Intent is referred to as *internal* or *external*, respectively.

In addition, there are two ICC types in terms of linkage *specificity*. For an ICC, if the sending component explicitly specifies the name of the receiving component (in the `component` field of the associated Intent), the ICC is *explicit*. Otherwise, the Android framework will need to determine at runtime the receiving component according to the `action`, `category`, and `data` fields in the associated Intent set by the sending component, and the ICC is *implicit*. The associated Intent is *explicit* or *implicit*, accordingly.

Intent resolution and ICC mapping. Intent-based ICC calls are both asynchronous and run-time binding within the Android framework. *Intent resolution* refers to the analysis for determining the value of each field of an Intent object. Based on the results of Intent resolution, the component sending an Intent may be matched to an appropriate receiving component capable of handling whatever operation the sending component is requesting as specified in the Intent. This matching process is referred to as *ICC mapping*.

With an explicit Intent, since it specifically has the target (receiving) component set, the Android platform simply uses the specified target component to map the corresponding ICC.

With an implicit Intent, however, the target component is uncertain until runtime, when the Android platform uses the three Intent fields mentioned above to match the same fields of the *Intent filters* specified for each component in the manifest file of the app. These filters claim what kinds of Intents (in terms of the three fields) the associated components can handle. If the matching succeeds, the latter component will receive and handle the Intent.

III. Visual ICC Dissection

We first give an overview of our holistic approach for ICC dissection, and then present our solutions to ICC data analysis and multiple-view, interactive visualizations.

A. Overview

Figure 1 depicts an overview of our technical approach (left), which contains two packages: *instrumentation* and *graphing*, and an outlook of the user interface (right), which consists of three views: *dynamic ICC view*, *static ICC view*, and *dashboard*. The interface outlook also illustrates the workings of our approach in a real use scenario. The two packages are decoupled for extensibility such that (1) the instrumentation package may provide streaming data for varied post-processing steps while (2) the graphing package may take different streaming sources (of graph structures) for interactive, visual examination. Moreover, the (inter-process) communication between these two packages is realized via TCP socket such that the user may use this approach with an actual device in hand, as opposed to relying on an emulator.

Given the APK of an app under analysis, the first package instruments it for tracing dynamic ICC calls. Later the instrumented APK is installed to and manipulated by the app user on a device, when the resulting traces are streamed to the second package for visualization. The core of the visualization is graphing both static and dynamic ICC information, which automatically updates upon user inputs to the running app. The user can then simultaneously navigate the app and explore the visualization for dissecting the app construction and behaviors, both based on the ICC visualizations and by referring to the dashboard where run-time statistics about the ICCs exercised (in addition to graph legends) are displayed.

B. ICC Data Analysis

To enable dissecting ICCs in an app, the ICC data must be computed before representing it for visual exploration. We compute two main categories of ICC data: *static* ICC mapping, and *dynamic* ICC traces. Mapping all ICCs based on the app code requires (1) static ICC analysis that resolves the ICC Intent at all ICC invocation sites (e.g., `startActivity`) and (2) parsing the manifest file, followed by (3) the matching between ICC Intents and Intent filters, as described earlier (Section II). In particular for (1), we utilized the most precise existing ICC resolution technique [11]. The other two steps are straightforward using standard methods [17]. The static ICC data is then represented as a component-level call graph, referred to as *static ICC graph*, where all possible ICC calls are considered. As expected, this static call graph is imprecise due to the imprecision of Intent resolution and the conservativeness of static ICC mapping (for safety).

The dynamic ICC traces are generated through the APK instrumentation. This data also is represented as a component-level call graph, albeit only components exercised and com-

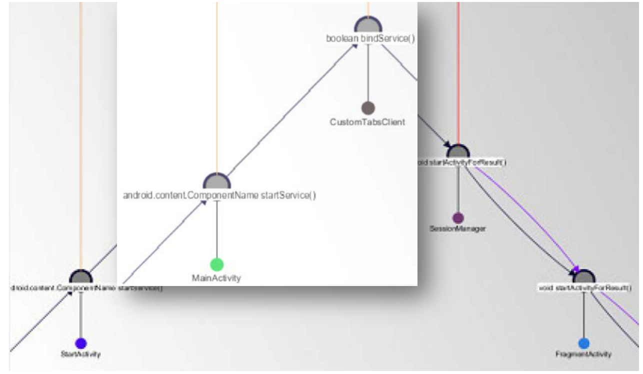

Fig. 2: A look into the dynamic ICC view with a closeup.

municating with at least one other exercised component are considered, hence the graph is referred to as a *dynamic ICC graph*. Notably, the dynamic ICC traces are produced as *streaming data* to enable real-time visual explorations of ICCs as they are triggered by user inputs. Our dynamic ICC analysis is *online*, thus these traces are not stored to disk files but transmitted and visualized on the fly.

In both graphical ICC representations, content and type of associated Intents, as well as ICC types are annotated on the graphs to assist with ICC comprehension. Additionally, on the dynamic ICC graph, all ICC instances are encoded to depict the frequency of ICC calls.

C. Visual Exploration of ICCs

As shown in Figure 1 (right), the enabling solution to ICC dissection is an interactive, dynamic visualization of ICCs. The entire viewport consists of three *complementary* views that are semantically linked and enhanced by rich interaction features.

1) Multiple Linked Views: **Visual encodings.** Given that ICCs are essentially (implicit/explicit) component-level calls, (call) graphs are a natural visual representation of ICCs. Moreover, we use different shape and color encodings to represent various properties of an ICC. Specifically, as shown in the dashboard view of Figure 1, lines with differentiating colors indicate ICC types, and ICC sending and receiving sites are indicated by squares and spheres with different colors (red and gray), respectively. Components receiving *external* Intents are indicated by a square shape with blue color, while (caller) components sending *internal* Intents by spheres with a (non-gray) color palette applied. These encoding schemes are adopted in order to enhance visual perception of rich ICC information in the visualizations. Arrowed lines indicate the time ordering of ICC calls in the dynamic view, facilitating navigation along the time line of app executions. The graphs are depicted in a zigzag structure to maximally reduce clutters.

Dynamic ICC view. Traditional ICC analysis relies on static analysis in order to map ICC within an app, which limits app understanding by the inherent shortcoming of being forced to introduce assumptions into the results. For example, a static analysis approach concerning ICCs will not be able to definitively determine what components will invoke ICC at a given time, and so the analysis approach must (over-) conservatively match ICC sending sites with *potential* receiving components. In contrast, the dynamic ICC view as proposed is inherently more *intuitive* and *precise* in explaining how an app behaves at component level, as no assumptions need to be made about

which Intent is being resolved to which component at what point in an app's traversal.

This view holds the dynamic ICC graph, which has two kinds of nodes: (1) each *component node* representing an exercised component, and (2) each *ICC node* representing an exercised Intent. Every ICC node has as its label the specific callsite that invokes the associated Intent, and the caller is indicated by the component node that is attached to the ICC node. The graph has three kinds of edges: (1) each *ICC edge* connecting the two ICC nodes of the ICC link, with the edge label showing detailed ICC information (type and sending/receiving sites), (2) a second, *ICC time edge* between two ICC nodes indicating the time lapse between their generation in the graph, with the edge label giving the time amount which informs the user of how long the ICC has taken, and (3) the *attachment edge* linking an ICC node to the attached component node.

Figure 2 captures part of an example dynamic ICC view, where the `MainActivity` component initiates an ICC through a call to an ICC API `startService` and later another component `CustomTabsClient` invokes `bindService` for sending another ICC. Both ICCs are explicit and external in this case. In contrast to the dynamic view of Figure 1 showing ICCs received in an app, this figure depicts an app's sending ICCs. As the user manipulates the app, the dynamic ICC graph will be updated to add new ICCs, enabling visual tracking and comprehension of ICCs as they are being exercised. Note that the entire graph is plotted in a *chronological hierarchy* to visualize the ICCs as time-series data while reducing visual clutters.

Static ICC view. The dynamic view visualizes ICCs that are exercised by the user inputs, which may not trigger all possible ICCs in the app. We thus also provide a static ICC view, where the static ICC graph of the app is visualized (using similar edge/node notions as in the dynamic ICC graph). This static view serves as a necessary *context* for the dynamic view by showing unexercised ICCs as well, and how they are connected to those exercised ones. While conservative hence imprecise, the static call graph provides a holistic picture of the entire spectrum of ICC-related *app structure/construction*, complementing the dynamic view which highlights the run-time *app behaviors* externalized by the exercised subset of all potential ICCs. When ICC traces are streamed to the dynamic view, exercised ICCs are also marked with color in the static view, hinting the user about the part of the entire spectrum that has been explored thus far.

ICC dashboard. The two *graphical* views are further complemented by the dashboard view, where *tabular* information is displayed showing both graph legends (as described in visual encodings above) and ICC statistics (bottom right of Figure 1). This view consists of two tables, showing dynamic ICC summaries with respect to Intents and caller components, respectively. The Intent table (top of the dashboard view) shows the breakdown of ICCs over different categories (sending/receiving, explicit/implicit, and internal/external) with for each category the total number of instances and percentages (against all static ICCs) that are exercised so far as shown in the dynamic view. The caller table (bottom) shows the same statistics for each exercised caller component. These numbers offer a quick overview of the ICC dynamics in the app.

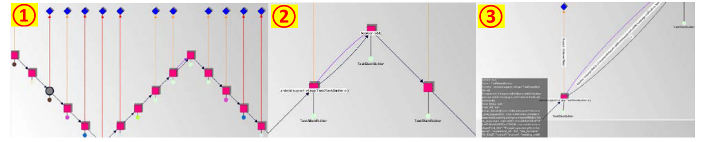

Fig. 3: Multiple continuous levels of details of ICC views.

2) In-situ User Interactions: Both the static and dynamic ICC graphs are *interactive*, allowing the user to control how to explore the ICCs. Moreover, the user interactions are designed as *in-situ* operations: visual aids/features become available right in place where the operations start. Currently, there are main categories of two in-situ interaction features.

First, hovering the mouse over a particular ICC node will generate a window containing every individual field of the captured Intent (as shown by the gray popup in the dynamic view of Figure 3). Note that not every Intent field is generally defined by the developer, so some fields will inherently be blank, unknown, or null.

Second, clicking on a component node will generate clue links between it and every other components of the same type (e.g., *Activity*) within the current dynamic ICC graph. These clue links can be clicked on or off at will, assisting the user in quickly grasping the frequency and pattern of specific component-level callers across the four component types in the running app (Section II).

3) On-demand Continuous Scaling: A notable interaction feature is that the user may continuously scale the two ICC views on demand. At different zoom levels, different levels of details will be visualized. Figure 3 depicts three example levels of details of the dynamic view, as numbered in ascending order of detail levels. The first level shows only symbolic graph nodes and edges. After zooming in appropriately, the view gets into the second level of detail, showing node annotations and additional edges (e.g., *ICC time edge*). Zooming in further will get into detail level 3 revealing edge labels, where Intent popups can be clearly seen upon in-situ triggers. When in finer detail levels, view sliders will appear allowing users to navigate the graphs via scrolling, and panning becomes active.

IV. Evaluation

We implemented a tool prototype of our approach and applied it to 16 benchmarks. The preliminary evaluation aimed at assessing the feasibility, usefulness, and practicality of the approach. For that purpose, all benchmarks are the latest top popular apps from Google Play. We instrumented each chosen app and manually traversed for a few minutes after installing it to an emulator. Next, we attempted to dissect the app behaviors using our tool while continuing to manipulate the app.

In an interesting revelation, the vast majority of ICCs occurring within a few of the studied apps were dedicated exclusively to advertising services. For example, upon opening the popular game app Sonic Dash, it was observed that over 68% of all the messaging occurring within the app was actually advertisement services setting up to run in the background. Specifically, 82 unique advertising ICC callsites were captured during the app's five-second start-up period, before users would even have a chance to interact with the app. Looking into the advertising ICCs, it was observed that many advertisers appear to intentionally attempt to obfuscate their Intent field data. While our tool was still able to correctly map

and visualize the advertisement services' behaviors despite the obfuscation, this practice is concerning as attempting to hide Intent information indirectly creates potential security risks to these apps which are mostly known as benign.

It should be noted as well, however, that there were also apps that showed no evidence of obfuscation, in advertising services or other app components. In one instance, a video game app Boom Beach made only one external ICC call throughout the entire traversal of the app. This app does host advertising services, yet the ICCs for setting these services up were far less pervasive and resource intensive when compared to other gaming apps. The ICC information in our visualizations such as those showing how an Android app handles a particular type of component (e.g., background services) will potentially be helpful to Google Play users in selecting appropriate Android apps for download. For example, a statistical analysis of ICC could be composed in an app's description on the app store, detailing how the specific app handles a device's resources in managing its run-time behavior. An end user could then use the basic analysis in identifying apps that make excessive ICC setting up services that are of no visibility, nor direct use to the user, and thus avoid a potentially negative experience.

Our tool also has enabled new findings in regards to suspicious behaviors even in these top apps. In one such instance, we analyzed the behavior of an anti-virus app. This app claims to find several viruses on the user's device after an initiated scan, and attempts to redirect the user to an external payment service so that the user may pay to have the viruses subsequently removed by the app. A quick (thirty-second) analysis of the app with our tool clearly demonstrated that at no point was the app actually conducting a system scan. In fact, clicking the 'Start Scan' button was directly linked to an `Activity` pane indicating that viruses had been detected, and providing an external link for payment.

In other suspicious apps, it was also observed that external Intents made up the majority of the Intents at all ICC sending sites. From a vulnerability analysis perspective, using external Intents are a basic strategy for malicious inter-app ICC [16]. For example, there are many instances in which a suspicious app seeking appropriate user permission will escalate permissions by invoking a component in a separate app on the victim's device that contains the necessary permissions [3].

In all cases, our tool enabled in-depth examination of the app behaviors related to ICCs by visually conveying how each ICC was triggered and connected, as well as how long it lasted. Our study also shows that the proposed approach is scalable for practical use. Over the 16 benchmarks, on average the instrumentation took 1.5 minutes and caused less than 2% runtime slowdown. Constructing the static ICC graph took 14.5 minutes on average, which however is a one-time cost for a given app, as is the cost for instrumentation.

V. Related Work

To date, we have not been aware of prior works addressing Android app comprehension with a focus on ICCs. As discussed earlier (Section I), most existing works involving Android ICCs focus on Intent resolution [10], [11] or address ICC mapping [3], [7], [15]. Yet, the resultant information does not immediately help with understanding ICC-induced app behaviors, especially those exercised at runtime. Other works aimed at detecting security threats exploiting ICCs as the main attack surface [4], [5]. Relevant empirical studies [2], [16] included general characterization of ICCs, which complement to our approach in that they provide a *broad* view on ICC-related app traits while ours offer a means for *in-depth* inspection of app construction and run-time behaviors due to ICCs. Our approach also is complementary to security solutions that target large-scale vetting by enabling dissection of ICCs for deep investigation of security issues. Meanwhile, advances in ICC resolution and mapping can enhance our approach via improved precision and efficiency of the static ICC graph.

VI. Conclusion and Future Work

As ICCs are a major obstacle for Android app comprehension and security, understanding ICCs is an essential step for both developing and securing apps, which to date has not been well assisted. To fill this grap, we propose to combine static and dynamic ICC analysis and enable dissection of resulting ICC information via interactive visual explorations. Preliminary case studies already showed promising merits and usage practicality of this approach in both app understanding and vulnerability analysis. A next step is to assess the approach through user studies with formal comprehension and/or security questions and a diverse user group. We also plan to further enhance the visualizations with additional visual and interaction features (e.g., a view of code related to ICCs).

References

[1] I. D. C. I. Research, "Android dominating mobile market," http://www.idc.com/promo/smartphone-market-share/, 2016.

[2] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in Android," in *DSN*, 2012, pp. 1–12.

[3] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *CCS*, 2014, pp. 1329–1341.

[4] K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-Based malware detection on Android," *TIFS*, vol. 11, no. 6, pp. 1252–1264, 2016.

[5] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *ISSTA*, 2015, pp. 118–128.

[6] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, 2017.

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *MobiSys*, 2011, pp. 239–252.

[8] C. Marforio, H. Ritzdorf, A. Francillo, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *ACSAC*, 2012, pp. 51–60.

[9] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *IEEE Security & Privacy*, no. 1, pp. 50–57, 2009.

[10] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *USENIX Security*, 2013, pp. 543–558.

[11] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *ICSE*, 2015, pp. 77–88.

[12] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: statically vetting Android apps for component hijacking vulnerabilities," in *CCS*, 2012, pp. 229–240.

[13] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *SOAP*, 2014, pp. 1–6.

[14] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Unified dynamic detection of Android malware," Tech. Rep. TR-17-01, January 2017, http://hdl.handle.net/10919/77523.

[15] D. Octeau, S. Jha, M. Dering, P. Mcdaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis," in *POPL*, 2016, pp. 469–484.

[16] H. Cai and B. Ryder, "Understanding android application programming and security: A dynamic study," in *ICSME*, 2017.

[17] Google, "Android Developer Reference: Intent ," http://developer.android.com/reference/android/content/Intent.html, 2017.