Modeling and Test Case Generation of Inter-Component Communication in Android

Ajay Kumar Jha, Sunghee Lee, Woo Jin Lee
School of Computer Science and Engineering
Kyungpook National University
Daegu, Republic of Korea
ajaykjha123@yahoo.com, lee3229910@gmail.com, woojin@knu.ac.kr

***Abstract*—Currently, there is a lack of tools or techniques which can clearly handle the complexity related to inter-component communication while developing Android applications. We propose a conceptual model which represents the inter-component communication at a higher abstraction level. We also propose a technique to derive test case from the model. The model can be useful in handling complexity at various stages of software engineering process. Mainly, it can be used for testing and analysis of inter-component communication in Android applications which we have demonstrated through experiment.**

***Index Terms*—Android, inter-component communication, modeling, test case generation, security analysis.**

I. INTRODUCTION

In Android applications there are two kinds of communication. First, the components interact with each other within the application which is called intra-application communication and second, one application communicates with another application which is called inter-application communication. Even in inter-application communication, it's the component of one application which communicates with the component of another application. In this paper, we will refer both types of communication as inter-component communication (ICC) unless specified.

There is a lack of existing tools and techniques which can practically model the ICC in Android applications. There may be several reasons behind this but in author's viewpoint the one reason which stands above all is the current practice followed during mobile application development. Most of the mobile applications are small to medium size in terms of source code and these applications are generally developed by either individual developer or very small team of developers without following any formal development process. Another reason may be time frame. Mobile applications are generally delivered within very short time duration in comparison to desktop applications.

However, as mobile applications become more complex, it is essential to follow standard software engineering processes to assure the development of secure, high quality mobile applications [1]. It has been proven in desktop applications that the model-based techniques greatly reduce the complexity during the software engineering process. For example, model-based testing approach improves the efficiency of testing procedures and helps in reusing the generated test cases. Overall, current trends of mobile application development process needs to be changed to produce high quality, secure applications.

In this paper, we present a conceptual model to represent the ICC in Android applications. One of the benefits of conceptual modeling is to help the stakeholders better understand a specific real-world domain and enhance communication among them [2]. In-line with this benefit, the main purpose of this work is to reduce the complexity of testing and analyzing ICC by better understanding and representing the domain. Though the proposed model can be either designed from requirement specification document or extracted from source code, we have developed a tool named ICCMATT (ICC Modeling And Testing Tool) which extracts the ICC graph from source code. Besides extracting the ICC graph, the tool also automatically generates test cases from the graph.

The rest of the paper is organized as follows. Section II describes background and related works. Section III describes the modeling concept in detail. Section IV describes the test case generation technique. Section V presents and evaluates the tool, ICCMATT and section VI concludes the paper.

II. BACKGROUND AND RELATED WORKS

A. Inter-Component Communication

Android applications are built using four types of components: activities, services, broadcast receivers, and content providers. These components are loosely coupled with each other and they are completely equipped to perform a single task independently but as the task becomes complex they interact with each other to complete the task. These components interact with each other through intent. Intent is an abstract description of an operation to be performed. Intent can be categorized mainly into two types: explicit and implicit. Explicit intent specifies the target component name whereas implicit intent does not specify any target component name instead it specifies the desired operation.

To receive intents, a component must be declared in the manifest file which is a configuration file in the application. Only broadcast receiver component can be declared either in the manifest file or during runtime. Manifest file is the single-most important resource for analyzing communication behavior of components. Through this file, a developer specifies whether the component will be able to communicate with component of another application and also whether a component can

communicate explicitly or implicitly. If a component is declared with exported flag set to true then the component can communicate with another application otherwise it can only communicate within the application. The flag's default value depends on the presence or absence of intent filter. If there is even a single intent filter then the component is exposed to another application.

B. Related Works

TaintDroid [3] tracks the privacy-sensitive data in the system. It does so by tainting or labeling sensitive data and then logging those data during application execution. TaintDroid raises the flag if any sensitive data outflows from the application. ComDroid [4] detects application communication vulnerabilities by statically analyzing components and intents. DroidChecker [5] uses inter-procedural control flow graph searching and static taint checking to detect exploitable data paths. JarJarBinks [6] tests the robustness of ICC by using fault injection technique. EPICC [7] finds vulnerabilities in ICC by connecting components, both within single applications and between different applications.

Some initiations have been taken towards model-based approach in engineering mobile applications, for example, in [8] authors extend the UML to represent specific features of Android applications and in [9] authors proposed model-driven approach to develop mobile applications. Techniques in [10] and [11], which focus on security aspect of Android applications, address the issue of ICC through formal modeling. Model-based conformance testing framework has been presented in [12]. In contrast to aforementioned formal model-based techniques for analyzing and testing ICC, our novel approach uses conceptual modeling with graphical representation.

III. MODELING CONCEPT

A. Component Representation

In our model, Android application components are represented by rectangle but the annotations in the rectangle differentiate the component types as shown in Fig. 1. We have used <*componentType: componentName*> annotation format which denotes the component type and its name respectively.

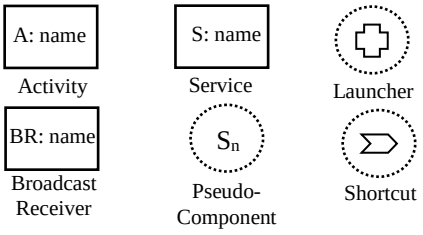

Fig. 1. Component representation

In certain situations during ICC the component's name and type are not known. In such situations we represent the component by pseudo-component as shown in Fig. 1. We name these pseudo-components uniquely as S_n where n is a non-zero natural number. Android applications may contain multiple entry points which are declared as launcher or shortcut in the manifest file. We will represent these entry points with special nodes as shown in Fig. 1.

B. Modeling Explicit Communication

We refer explicit communication to the communication among components which happens through explicit intent. Here, we have two communicating entities which in our case are components and a medium through which components communicate which is intent. We will refer the component which passes the intent as source and component which receives the intent as sink. The association between source and sink component will be represented by a directed edge from source to sink component.

A component may communicate with several other components through different intents. It's important to identify all the intents uniquely and incorporate into the model. We incorporate the intent by placing intent id on the edge between source and sink components. The intent id is denoted by I_n where n is a non-zero natural number. The communication between source activity a1 and sink activity a2 in Fig. 2 represent the explicit communication.

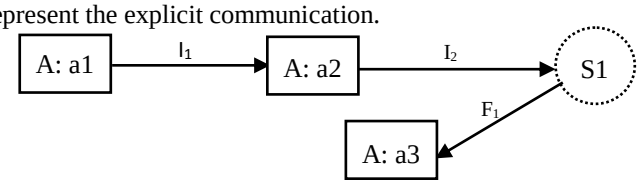

Fig. 2. Explicit and implicit communication

C. Modeling Implicit Communication

We refer implicit communication to the communication among components which happens through implicit intent. Similar to explicit communication, implicit communication has two communicating entities that are two components and a medium that is intent but unlike explicit intent, implicit intent does not define sink component. Sink component is determined dynamically by intent resolver using matching technique. Intent resolver first examines the intent fields and then it finds the sink component by matching the intent fields with the attributes of the intent filter bound to possible sink components. Altogether we have four entities in implicit communication: source and sink components, implicit intent and intent filter. Further we will represent and define the relationship among these entities through our model.

While establishing implicit communication with third party application by sending implicit intent, developers don't have any knowledge of third party application such as component name and intent filter. Developer just creates an implicit intent and then sends that implicit intent in the wild. Though it's possible to know the applications which can handle the specified operation, there is not any possible way to know the name of the sink component. In our model, we will represent this sink component by pseudo-component. The communication between source activity a2 and sink component S1 in Fig. 2 represents this implicit communication event.

There is another kind of event in implicit communication that is an application receiving implicit intent sent by third party application. Intent filter plays a crucial role in determining whether to receive or reject the implicit intent sent by third party application. There is no other way for third party application to enter into the application without going through intent filter. In this case, the implicit intent is unknown since it

originates from third party application but to accept the communication one of the intent filter must declare the same attributes as declared in the implicit intent. Instead of representing the edge by implicit intent, here it will be represented by intent filter F_n as shown by the edge between S1 and a3 in Fig. 2.

While receiving implicit intent from third party application, source as well as sink components are unknown because implicit intent originates from third party application and it does not declare any sink component. Here we will represent the source component by pseudo-component and we will place one pseudo-component for each intent filter declared in the application because each intent filter represents a possible entry point. Possible sink component can be determined by checking the component against which the intent filter has been declared. The communication between source component S1 and sink activity a3 in Fig. 2 represents this implicit event.

D. Modeling Pending Intent

Other than explicit and implicit intents, there is pending intent in Android. Pending intent acts as a wrapper for either explicit or implicit intents. After an application creates a pending intent, it is handed to third party application which later performs pre-defined task. It is most-widely used for notification and alarm services. Here we have two parts, first the application sends pending intent to third party application and second, third party application executes pre-defined task on the source application through wrapped intent. To simplify the model; we will merge these two parts into single construct as shown in Fig. 3. Here, the source component will be represented by the component which sends the pending intent and the sink component will be represented by the component which receives the intent wrapped inside the pending intent. The edge will be represented by the pending intent P_n.

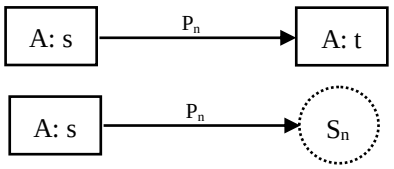

Fig. 3. Explicit and implicit communication through pending intent

E. Embedded Context Graph

This section can be taken as an extension to our core model. Intents are delivered from source to sink component through method call as a parameter. Here, context refers to the method's calling context that is the method in which intent delivery method has been called. For all kinds of communication, context is represented by edge between source component and sink component. For example if the context is *onClick()* method in explicit communication then it will be represented as shown in Fig. 4.

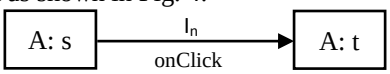

Fig. 4. Explicit communication with context

IV. TEST CASE GENERATION

The main goal of testing ICC is how the component behaves after receiving intents that is whether the component behaves normally or it shows any unexpected behavior. It is not that only the sink component will be affected by receiving the intent because the same sink component may act as a source component in the same application and the effect caused by intent will propagate to other components. So it's important to tap the propagation while testing ICC. Our aim is to obtain sequence of end-to-end ICC events. End-to-end means from one of the entry points to one of the exit points of the graph. Each such test sequence provides an executable test case.

The ICC graph generated through our technique has several properties. First, the graph is multigraph with parallel edges. Second, within single graph there are several entry and exit nodes. Third, the graph may be cyclic. And fourth, the graph may contain loop (pseudograph). Considering all these properties we have proposed a novel test case generation algorithm which is shown in Fig. 5. We have also proposed intent and intent filter (I-IF) coverage criteria which can be collectively used to obtain adequate and effective test suite from our proposed model. According to our coverage criteria each intent and intent filter should be covered at least once.

```
1:  Input: ICCGraph G
2:  Output: Test Case File F
3:  Declare ArrayList: entryNodes, exitNodes, edgeList, visitedEdgeList, testCase
4:  add(entryNodes, nodes which do not have incoming edge in G)
5:  add(exitNodes, nodes which do not have outgoing edge in G)
6:  add(edgeList, all the edges in G)
7:  while size(visitedEdgeList) != size(edgeList)
8:      foreach element in entryNodes
9:          Declare String: chosenEdge, sourceNode
10:         chosenEdge <- null
11:         testCase = empty
12:         sourceNode <- element of entryNodes
13:         add(testCase, sourceNode)
14:         while exitNodes does not contain sourceNode
15:             Declare ArrayList: outEdges
16:             add(outEdges, all the outgoing edges of sourceNode)
17:             foreach element in outEdges
18:                 if visitedEdgeList does not contain the element of outEdges
19:                     chosenEdge <- element of outEdge
20:                 end if
21:             end foreach
22:             if visitedEdgeList contains all the edges in outEdge
23:                 if testCase contains all the edges of outEdge
24:                     chosenEdge <- one random edge from outEdge
25:                 else
26:                     chosenEdge <- random edge from outEdge which is not in testCase
27:                 end if
28:             end if
29:             if chosenEdge is not null
30:                 if immediate predecessor edge in testCase is same as chosenEdge
31.                  and outEdges has single element
32:                     add(exitNodes, target node of chosenEdge)
33:                 else
34:                     append(testCase, chosenEdge)
35:                     append(testCase, target node of chosenEdge)
36:                     sourceNode <- target node of chosenEdge
37:                     if chosenEdge is not in visitedEdgeList
38:                         add(visitedEdgeList, chosenEdge)
39:                     end if
40:                 end if
41:             end if
42:         end while
43:         if testCase contains at least one unvisited edge
44:             add(F, testCase)
45:         end if
46:     end foreach
47: end while
```

Fig. 5. Test case generation algorithm

V. ICCMATT

ICCMATT is a completely automated Eclipse plug-in tool written in Java. The high-level design of ICCMATT is shown in Fig. 6. It takes application's source code as input and produces two different files as output: a text file containing all the test cases and a GraphML file containing all the application specific data to view ICC graph.

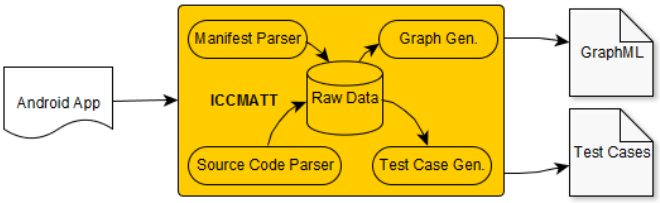

Fig. 6. High-level design of ICCMATT

We evaluated efficiency and effectiveness of ICCMATT tool on five Android applications: connectbot, K-9 Mail, opensudoku, Tomdroid, and Avare. The ICC graph generated by our tool for K-9 Mail is shown in Fig. 7. Number of test case generated, I-IF coverage, and execution time of ICCMATT is shown in table 1. Data in table 1 clearly show that the ICCMATT tool is effective as well as efficient in extracting ICC graph and generating test cases. The generated ICC graph can not only be used for generating test cases but also be used for various software engineering purposes including security analysis. For example, we can easily identify entry and exit components which are vulnerable for malicious data injection and privacy leak respectively. The last column of table 1 shows the number of security vulnerable components in our benchmark applications.

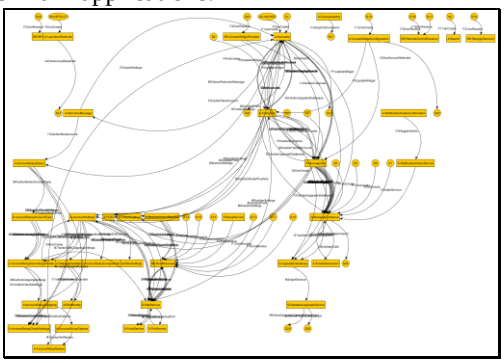

Fig. 7. ICC graph of K-9 Mail app

Table 1. Evaluation results

App. Name	Execution Time (ms)	Test Case	I-IF Coverage (%)	Security Risk Components
K-9 Mail	67578	58	89	17
connectbot	1131	13	78	4
opensudoku	954	8	81	5
Tomdroid	1154	12	82	4
Avare	1895	24	72	3

VI. CONCLUSION

In this paper, we have proposed modeling and test case generation technique for ICC in Android applications. We have also developed a tool to automatically generate ICC graph and test cases from the application's source code. Evaluation results show that the tools can be effectively and efficiently used for testing and security analysis of Android applications. We further need to evaluate the tool on large scale benchmark applications with some extension towards automated security analysis which includes addition of Android permission model and security report generation.

ACKNOWLEDGMENT

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. NRF-2014R1A1A2058733) and the IT R&D program of MSIP/IITP [10041145, Self-Organized Software platform (SoSp) for Welfare Devices].

REFERENCES

[1] Anthony I Wasserman, "Software engineering issues for mobile application development", in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 397-400.

[2] Peter P. Chen, Bernhard Thalheim, and Leah Y. Wong, "Future directions of conceptual modeling", in *Conceptual modeling*, 1999, pp. 287-301.

[3] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones", in *Communications of the ACM*, 57(3), 2014, pp. 99-106.

[4] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android", in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011, pp. 239-252.

[5] P.P. Chan, L. C. Hui, and S. M. Yiu, "Droidchecker: analyzing android applications for capability leak", in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012, pp. 125-136.

[6] 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 *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012, pp. 1-12.

[7] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis", in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[8] M. Ko, Y. J. Seo, B. K. Min, S. Kuk, and H. S. Kim, "Extending UML Meta-model for Android Application", in *Proceedings of the IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS)*, 2012, pp. 669-674.

[9] F. T. Balagtas-Fernandez, and H. Hussmann, "Model-driven development of mobile applications", in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pp. 509-512.

[10] A. Armando, G. Costa, and A. Merlo, "Formal modeling and reasoning about the Android security framework", in *Proceedings of the 7th International Symposium on Trustworthy Global Computing*, 2013, pp. 64-81.

[11] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing Android's permission system", in *Computer Security–ESORICS*, 2012, pp. 1-18.

[12] Y. Jing, G. J. Ahn, and H. Hu, "Model-based conformance testing for android", in *Proceedings of the 7th International Workshop on Security (IWSEC)*, 2012, pp. 1-18.