ICCMATT: Modeling, analysis, and test case generation of inter-component communication in Android

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

Abstract: Android applications are composed of components that may interact with each other to accomplish a task. Moreover, a component of one application can also communicate with a component of another application. Inter-component communication is an integral part of Android applications. Incorrect implementation of inter-component communication can cause serious reliability and security issues. This paper presents a conceptual model that represents the inter-component communication at a higher abstraction level. It also presents a completely automated tool that extracts the model from the source code. The extracted model can be used for analyzing inter-component communication at a higher abstraction level. The tool also generates executable test cases from the extracted model. Finally, the tool generates a security report indicating vulnerable inter-component communication. Overall, the tool assists developers in mitigating reliability and security issues caused by errors in implementing inter-component communication.

Keywords: Android applications, inter-component communication, modeling, test case generation, security analysis

1. Introduction

An Android application can utilize the functionalities or services offered by other applications while still giving the impression of a single, seamless application, which means developers don't have to develop all the functionalities by themselves. Instead, they can use a large number of services offered by other applications. For example, if a developer needs to provide a map-based service in her restaurant application then she can utilize the service offered by the Google map application. The approach provides flexibility in developing applications by utilizing existing services. However, it enhances the complexity of developing an application in the form of inter-application communication, which may not only result in error prone applications but also result in high-security risk applications [1, 2, 3, 4]. Our previous studies [5, 6] have identified several

developer mistakes in implementing inter-application communication that result in serious reliability and security issues.

Android applications are composed of components. A component-based system has both advantages and disadvantages. While it's easier to develop a component, it's extremely difficult to integrate the components. Moreover, the components in an application may interact with each other to accomplish a task, which significantly increases the complexity in designing interface for intra-application communication. Although most of the Android applications are small in size with less than 10 components, the Google Play store has a significant number of large applications. Our empirical study [5] on 13,944 applications from the Play store shows that 49% of the applications have more than 10 components. Furthermore, 17 of the applications have more than 200 components each. Implementing intra-application communication for a large number of components is a cumbersome and error prone task, for example, the bug **#84** in Yalp Store application [7] and the bug **#13** in Pinwheel Messenger application [8] cause the applications to crash. These reported bugs are caused by starting an activity component from outside of an activity context without setting the FLAG_ACTIVITY_NEW_TASK flag.

While the errors in implementing inter-application communication result in both reliability and security issues, the errors in implementing intra-application communication result in only reliability issues. In this paper, we refer both inter-application and intra-application communication as inter-component communication (ICC) unless specified. The complexity of ICC that includes state and event management of components and lack of expertise in developers are the leading causes of reliability and security issues in Android applications [9, 10, 11]. A solution is to develop and adopt automated software engineering practices for application development to produce secure, high-quality mobile applications [12], especially model-based software engineering techniques can significantly reduce the complexity. However, regardless of the recent advancement in tools and techniques for testing Android applications [13], there is a lack of tools and techniques for testing ICC in Android 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 [14]. 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. Although the model can be either designed from requirement specification document or extracted from source code, we have developed an open source Eclipse plugin tool named ICCMATT (**ICC** **M**odeling, **A**nalysis, and **T**esting **T**ool) [15] that extracts an ICC graph from the source code. The extracted ICC graph provides a complete picture of intra-application and inter-application communication performed by an application. Developers can find ICC vulnerabilities, particularly unprotected exported components and communications that may leak sensitive data, by looking at the ICC graph. Besides extracting the ICC graph, the tool automatically generates executable test cases and a security report. We have developed an algorithm that generates test cases from the ICC graph. The generated test cases can be executed to identify reliability issues caused by errors in implementing ICC. The security report provides information on two kinds of vulnerabilities: the name of the exposed components that

may need permission-based protection and the inter-application communication that may leak sensitive data. A preliminary version of this work has been published in a short paper [16].

The main contributions of this paper can be summarized as follows:

- Presents a novel conceptual model for inter-component communication in Android.
- Presents a test case generation algorithm that generates test cases from the model and presents a security analysis algorithm that reports vulnerable ICC.
- Presents an open source tool called ICCMATT that automatically extracts an ICC graph from the source code. The tool also generates executable test cases from the extracted graph. Moreover, the tool generates a security report indicating vulnerable ICC.
- Evaluates efficiency and effectiveness of the ICCMATT on 90 active and open source Android applications.

The rest of the paper is organized as follows. Section 2 briefly describes the background on ICC and security in Android applications. It also presents related works. Section 3 presents the conceptual modeling of inter-component communication. Section 4 and 5 present the security analysis algorithm and the test case generation technique, respectively. Section 6 presents the implementation of the ICCMATT. The evaluation results of ICCMATT and threats to validity are discussed in Section 7. Finally, the paper concludes in Section 8 with future directions.

2. Background and related works

2.1 Inter-component communication

Android applications are built using four types of components: activities, services, broadcast receivers, and content providers. An activity represents a single user screen with which users interact. Services are generally used to perform long-running background tasks with no interaction from the users. Broadcast receivers handle the system-wide broadcast events that may originate from the system or applications. Content providers manage access to the database through content URI that uniquely identifies its data set.

The components of an application may interact with each other to accomplish a task. The interaction is performed through a message passing technique using an intent. Except for the content provider components, all other components interact using an intent, which is an abstract description of an operation to be performed. The intent encapsulates various optional and mandatory attributes such as a target component name, an action to be performed, data to be acted on, a category describing additional information about the target component, key-value pairs that carry extra data, and flags functioning as metadata for the intent [17]. Intents can be categorized mainly into explicit and implicit intents. An explicit intent specifies a target component name that receives the intent, whereas an implicit intent does not specify a target component name. Instead, it specifies the desired operation. Android selects the target component that supports the desired operation during run-time [17]. There is one more type of intent called pending intents that act as a wrapper for explicit or implicit intents. The pending intents are used to perform an operation at

a future event. An application creates a pending intent and hands it over to another application that later performs the predefined task on the source application. It is most-widely used for notification and alarm services.

The components of an application are declared in an Android manifest file, which is a configuration file in the application. Broadcast receiver components can also be declared in the source code. The manifest file is the most important resource for analyzing communication behavior of components. Developers specify communication behavior of a component by setting its various attributes in the manifest file. For example, if a component is declared with exported flag set to true then the component can perform inter-application communication otherwise it can only perform intra-application communication. The flag's default value depends on the presence or absence of an intent filter. An intent filter [17] advertises capability of a component to handle an implicit intent. If there is even a single intent filter then the component is exposed to another application otherwise it is confined within the same application. The intent filter also determines whether the component can communicate explicitly or implicitly. The presence of an intent filter specifies that the component can communicate implicitly otherwise only explicit communications are allowed. A component willing to accept an implicit intent must declare an intent filter with the same attributes as declared in the implicit intent. More details on inter-component communications can be found in [1].

2.2 Security in Android

Security in Android is implemented at both system and application levels. At the system level, Android uses the sandboxing technique, which creates a virtual isolated execution environment. The resources required to execute an application, for example, Dalvik Virtual Machine [18], file system, memory, etc. are granted exclusively to the application. The approach largely prevents malicious activities. However, at the application level, an Android application sometimes needs to make holes in the isolation wall created by the sandboxing technique to communicate with the outside world through different mediums such as network, shared file, etc. It may also communicate with other Android applications through inter-application communication mechanism using intents. These communications create holes in the isolation wall that are exploited by malicious applications to perform malicious activities on benign applications. Thus, regardless of the existence of the sandboxing technique at the system level, Android needs protection at the application level and it uses the permission-based security for that purpose.

In Android, a permission can be assumed as a label that is placed on a sensitive object. If an application needs to access the sensitive object then it has to acquire the label first. There are two types of sensitive objects in Android. The first one is shared resources such as contact address that is protected by system defined permissions. For example, if an application needs to access the contact address then it has to declare the "READ_CONTACTS" permission and users have to grant the permission to the application. The second type of sensitive object is an application's components. Android applications are built using components that can be exposed to third-party applications. The exposed components are highly vulnerable to attacks; therefore, they need to be protected with developer-defined custom permissions. An application willing to access the exposed components must declare the custom permissions. For example, if a component C of an

application A is protected with “MY_PERMISSION” permission then another application has to declare the same permission “MY_PERMISSION” in its manifest file to access the component C of the application A. Users must grant the permission to the application during install-time or runtime depending on the Android version. More details about Android security can be found in [19, 20, 21].

2.3 Related works

Inter-component communication in Android applications has been extensively studied for security analysis [1, 3, 10, 11, 22, 23, 24, 25, 26, 27, 28]. Most of the existing techniques in this stream focus on data and control flow analysis [1, 3, 22, 23, 24, 27] because Android applications may leak sensitive data during inter-application communication. TaintDroid [22] tracks privacy-sensitive data by tainting sensitive data and then logging the data during an application execution. It raises a flag if any sensitive data outflows from the application. However, TaintDroid does not support taint tracking on IPC that takes place through intents, which is the most frequently used IPC mechanism in Android. EPICC [24] finds vulnerabilities in ICC by connecting components both within an application and among applications. It extracts all the exit points of an application that can send intents then it determines the value of the intent by using string analysis and Interprocedural Distributive Environment (IDE) analysis. Finally, it determines the possible targets or entry points based on the value of the intent. A more precise version of EPICC has been presented in IC3 [25]. IC3 performs composite constant propagation to calculate the precise value of intent at exit points, which reduces the number of false positive. IccTA [3] detects privacy leak in an intent-based IPC mechanism by modifying the source code of the application. It leverages the existing tools EPICC [24] and Flowdroid [27] to perform the task. ComDroid [1] detects ICC vulnerabilities by analyzing components and intents. It generates a warning if a component is exposed and protected with no permission or weak permission. ComDroid also issues a warning when it detects an implicit intent being sent with weak or no permission. DroidChecker [23] uses inter-procedural control flow graph searching and static taint checking to detect exploitable data paths. Similar to ComDroid, it first checks for vulnerable components by examining the manifest file, which are then further analyzed for capability leak detection by using taint propagation in the interprocedural control flow graph. Although these techniques provide valuable information for closing security loopholes, the techniques work on very low-level abstraction. It’s often cumbersome to work on such a low-level abstraction unless the technique is completely automated. Even state of the art [22, 23, 27] in this category fails to completely serve its purpose due to the failure to capture Android specific inter-application communication mechanism effectively.

Meanwhile, there is a lack of model-based techniques that can analyze ICC in Android applications. Model-based techniques are helpful not only in analyzing at a higher level of abstraction but also during entire software engineering processes. Some initiations have been taken towards a model-based approach in engineering mobile applications. Ko et al. [29] extended the UML to represent specific features of Android applications, and Balagtas-Fernandez et al. [30] proposed a model-driven approach to develop mobile applications. Although these model-based approaches reduce the complexity in the development process, these techniques do not address the specific issues of ICC. However, formal model-based techniques [31, 32] that focus on the security

aspect of Android applications have been proposed. Armando et al. [31] introduced a framework for defining an application in terms of components, manifest, and namespace. Then, they presented a formal semantics for describing computations in the model. Fragkaki et al. [32] developed a framework to perform a formal analysis of Android-style permission. Although it mainly focuses on permission, it also addresses some of the issues of ICC. A model-based conformance testing framework has been presented in [33]. It first extracts a formal model of ICC and then generates test cases from the extracted model.

In contrast to existing techniques, ICCMATT resolves complexity by providing a graphical model of ICC. On top of the model, it provides testing and security analysis techniques. Besides ICCMATT tool, the paper provides a conceptual model of ICC that can be designed from a requirement specification document. The designed model can be used during entire application development process. The importance of this model may not be significant for small applications. However, for complex applications, the model can play a vital role in producing high-quality secure applications. While modeling and test case generation techniques including ICCMATT tool are novel contributions of this paper, the algorithm used for security analysis is similar to ComDroid [1]. However, our approach provides more information in the form of an ICC graph that assists developers in making correct security decisions.

3. Inter-component communication Modeling

Conceptual modeling is both art and science. Art in the sense how well the graphical representation of the model facilitates different kinds of communication among stakeholders and science in the sense how well the model can be utilized in different engineering processes. To excel in both art and science of conceptual modeling, one has to deeply understand the domain that the model represents.

3.1 Formal representation of the model

Although it is not necessary to formalize a conceptual model before designing, explicit formalization can avoid misunderstandings. It can also help in verifying the model. The proposed model is composed of nodes and edges in which components are represented with nodes while intent and intent filters are represented with edges.

DEFINITION: *The ICC model is represented with a 2-tuple (N, E), where N represents all the nodes; and E represents all the edges.* $N = Activities \cup Services \cup BR \cup TC \cup L \cup SC$, *where Activities is a set of activity components defined in the manifest file, Services is a set of service components defined in the manifest file, BR is a set of broadcast receiver components defined in the manifest file and declared dynamically in the source code, TC is a set of components from third-party applications, L is an application launcher, and SC is a set of shortcuts for the application.* $E = \langle s, t, i \rangle \mid \langle s, t, f \rangle$ *where* $i \in I$, $f \in F$, *s is a source component, t is a target component, i is an intent, f is an intent filter, I is a set of intents in the application, and F is a set of intent filters in the application.*

Based on the formal definition, we will build the conceptual model. Conceptual modeling specifies and describes the major design metaphors and analogies employed in the design, the concepts the system exposes to users, the relationships between these concepts and the mappings between the concepts and the task-domain [34]. We will explore these artifacts of conceptual modeling in the context of the small but important domain of ICC in Android applications.

3.2 Components representation

Components of an Android application are represented by rectangles in the ICC model. The annotations in the rectangles differentiate the component types as shown in Figure 1. We have used a <*componentType: componentName*> annotation format, which represents the component type and its name respectively. Furthermore, we have used short names for component types as A, S, and BR for activity, service and broadcast receiver respectively. In this paper, we present a conceptual model of ICC that takes place through intents. Content provider components do not interact through intents; therefore, Figure 1 does not include the graphical representation of the content provider component.

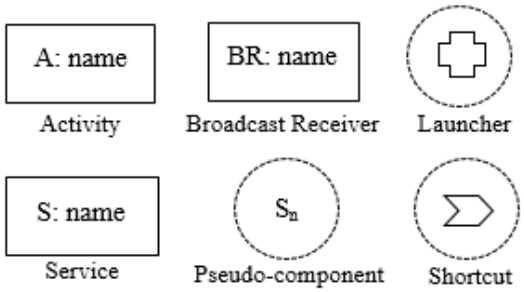

Figure 1. Components representation.

The component name and its type are undisclosed in some ICC for example, the target and the source component respectively are unknown when an application sends and receives implicit intents. We represent the unknown source and target components by pseudo-components as shown in Figure 1. Specifically, a pseudo-component represents a component of a third-party application that can be system resources, system applications, or user applications. The pseudo-components are identified uniquely as S_n where n is a non-zero natural number. Unlike Java-based applications, Android applications may contain multiple entry points, which are declared as a launcher or shortcuts in the manifest file. These entry points are represented with special nodes as shown in Figure 1. Logically, implicit communications also create entry points that are represented with pseudo-components.

In this paper, we categorize ICC into three different types based on the types of intents involved: explicit communication, implicit communication, and pending communication. Moreover, we illustrate the modeling concept through a running example shown in Figure 2. The code in the running example has been taken from an open source Android application named *AppAlarm*. For brevity, the figure shows only relevant code fragments. The running example has three components: an activity named *AlarmList*, a service named *AalService*, and a broadcast receiver named *SnoozeWakeupReceiver*. These three components communicate with each other through intents to perform certain tasks.

```
public class AlarmList extends ListActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
            Intent i = new Intent(this, AalService.class);
            i.setAction(AalService.ACTION_SET_ALARM);
            startService(i);
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        Intent i;
        switch (item.getItemId()) {
        case MENU_OTHER_APPS:
                i = new Intent(Intent.ACTION_VIEW);
                i.setData(Uri.parse(getString(R.string.e6_market_uri)));
                startActivity(i);
                break;
        }
        return super.onOptionsItemSelected(item);
   }
}

public class AalService extends Service {
        private void cancelSnoozeAlarm() {
                PendingIntent sender = PendingIntent.getBroadcast(this, 0, new
                Intent(this, SnoozeWakeupReceiver.class), 0);
                AlarmManager alarmManager = (AlarmManager)getSystemService(ALARM_SERVICE);
                alarmManager.cancel(sender);
                cancelAlarmSnoozeNotification();
        }
}

public class SnoozeWakeupReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context ctx, Intent arg1) {
                Intent i = new Intent(ctx, AalService.class);
                i.setAction(AalService.ACTION_RECOVER_SNOOZE_ALARM);
                ctx.startService(i);
        }
}
```

Figure 2. Running example from AppAlarm application.

3.3 Modeling an explicit communication

An explicit communication is a communication among components that takes place through an explicit intent. Modeling explicit communication is relatively a trivial task. Here, we have two communicating entities in the form of components and a medium in the form of an intent through which components communicate. The component that passes the intent is referred as a source component and the component that receives the intent is referred as a sink component. The association between a source and a sink component is represented by a directed edge from the source to the sink component.

A component may communicate with several other components through different intents containing different attributes. In the ICC model, each intent is assigned a unique identity I_n where n is a non-zero natural number. An intent is represented by a directed edge from a source to a sink component. The running example in Figure 2 has two explicit intents. One intent originates in *AlarmList* activity that starts *AalService* service and another originates in *SnoozeWakeupReceiver*

broadcast receiver that starts *AalService* service. The communication through intents I_1 and I_2 in Figure 3 represents explicit ICC model for the running example. It is important to mention that the communication within an application should be always explicit to avoid any security risk [17]. Modeling explicit communication may not be that much important for security analysis. However, it is important for other software engineering processes such as testing.

3.4 Modeling an implicit communication

An implicit communication is a communication among components that takes place through an implicit intent. Similar to the explicit communication, an implicit communication has two communicating entities that are components and a medium that is an intent. However, unlike an explicit intent, an implicit intent does not declare a sink component. It is determined dynamically by an intent resolver using a matching technique [17]. The intent resolver finds the sink components by matching the intent's attributes with the attributes of the intent filter bound to possible sink components. Altogether, there are four entities in an implicit communication: a source component, a sink component, an implicit intent, and an intent filter. We represent and define the relationship among these entities through the ICC model.

While establishing an implicit communication with a third-party application by sending an implicit intent, a developer doesn't have any knowledge of the third-party application such as component name and intent filter. A developer creates an implicit intent that encapsulates the operation to be performed and then sends the implicit intent in the wild. Although it's possible to identify the applications that can handle the specified operation, there are not any possible ways to identify the name of the sink component other than through reverse engineering the application. Here, the unknown sink component is represented by a pseudo-component in the ICC model. In the running example, *AlarmList* activity starts an activity implicitly. The communication through the intent I_3 in Figure 3 represents the implicit communication. The ICC model does not differentiate between an implicit and an explicit intent while assigning the unique identity. The identity of both explicit and implicit intents is treated as a global property in an application. However, the graphical representations clearly differentiate between the intents. The sink component of an implicit intent must be a pseudo-component as shown in Figure 3.

There is another kind of event in an implicit communication. An application can receive implicit intents sent by third-party applications. Here, both source and sink components are unknown because an implicit intent originates from a third-party application and it does not declare a sink component. However, each intent filter in an application represents a possible entry point for an implicit intent; therefore, a pseudo-component for each intent filter declared in the application represents a source component in the model. Although an implicit intent does not declare a sink component, probable sink components can be determined by checking the components against which the intent filter has been declared. In this kind of event, the association between the source and sink components is represented by an intent filter instead of an implicit intent. The intent filters are identified uniquely by F_n where n is a non-zero natural number. It is important to note here that the launcher and the shortcut discussed in Section 3.2 are declared through intent filters; therefore, these entities will be represented by replacing the pseudo-component with the launcher or the shortcut symbol.

Intent filters for activity and service components are declared in the manifest file whereas intent filters for broadcast receivers can be declared inside the manifest file and the source code; therefore, both the source code and the manifest file need to be analyzed to determine the events that receive implicit intents from third-party applications. A fragment of the manifest file declared by the *AppAlarm* application is shown in Figure 4. It has two intent filters declared against *AlarmList* activity component. The intent filter containing action as MAIN and category as LAUNCHER indicates that *AlarmList* is the main entry point of the application. Another intent filter indicates that *AlarmList* can accept implicit intents from third-party applications. Including these implicit communications in the running ICC model results into the model shown in Figure 3.

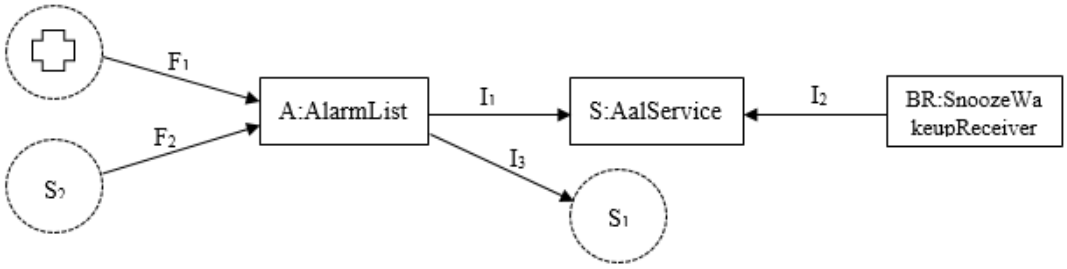

Figure 3. Modeling explicit and implicit ICC for the running example.

```
<activity android:name=".AlarmList"
     android:label="@string/app_name" android:configChanges="keyboard|keyboardHid-
     den|orientation">
       <intent-filter>
          <action android:name="android.intent.action.MAIN" />
             <category android:name="android.intent.category.LAUNCHER" />
       </intent-filter>
       <intent-filter>
          <action android:name="android.intent.action.VIEW"></action>
          <category android:name="android.intent.category.BROWSABLE"></category>
          <category android:name="android.intent.category.DEFAULT"></category>
          <data android:scheme="http" android:host="episode6.com" android:pathPre-
          fix="/ibuilder/m/"></data>
       </intent-filter>
</activity>
```

Figure 4. Fragment of the manifest file from AppAlarm application.

3.5 Modeling a pending communication

A pending communication is a communication among components that takes place through a pending intent. It can be clearly separated into two parts: an application sends a pending intent to a third-party application and the third-party application executes the predefined task on the source application through a wrapped explicit or implicit intent.

In the first part where an application sends a pending intent to a third-party application, a source component can be easily identified and the third-party application receiving the pending intent that acts as a sink component can also be easily identified. Here, the sink component is temporarily represented by a pseudo-node as shown in Figure 5(a). The pending intent is represented by a directed edge from a source to a sink component and it is identified uniquely by P_n where n is a non-zero natural number.

In the second part, the third-party application executes the predefined task through the intent wrapped inside the pending intent. The wrapped intent may be an explicit or an implicit intent. Here, the sink component of the first part becomes the source component. The directed edge is represented by a wrapped explicit or implicit intent. A sink component is represented by an actual component or a pseudo-component depending on the wrapped explicit or implicit intent respectively as shown in Figure 5(b). To simplify the ICC model, these two parts are merged into a single construct as shown in Figure 5(c). Here, the source component is represented by the component that sends the pending intent and the sink component is represented by the component that receives the intent wrapped inside the pending intent. The edge is represented by the pending intent. The running example has one pending intent in *AalService* component. Including the pending communication in the running ICC model results into the model shown in Figure 6.

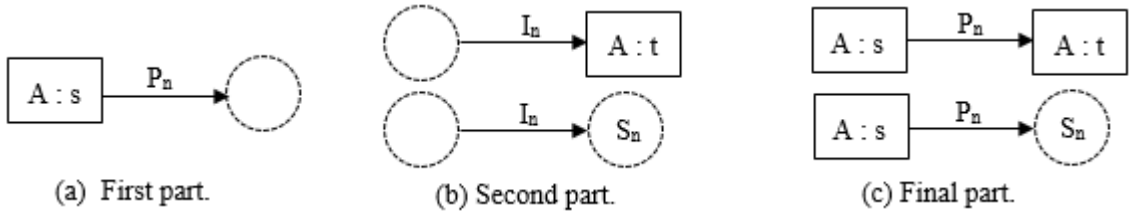

Figure 5. Modeling a pending communication.

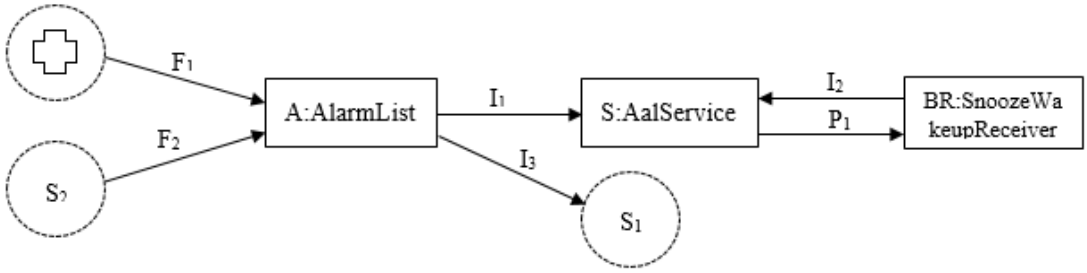

Figure 6. Modeling pending ICC for the running example.

The model in Figure 6 provides a complete picture of ICC for the running example. In addition to ICC, the model provides various information about the application and its security issues. One can observe that the example application has components: one activity, one service, and one broadcast receiver. *AlarmList* activity is the main component and it can receive intents from third-party applications; therefore, a developer should ascertain that there is no security risk. The *AlarmList* activity can also send an intent to third-party applications; therefore, a developer should check for privacy leaks. ICC in Android applications can be modeled using the four basic constructs presented in Sections 3.3, 3.4, and 3.5. We have tried to keep the conceptual ICC model as simple as possible while providing all the required functionalities, which is the basic requirement for any conceptual model [34].

3.6 Adding contextual information

Although the information incorporated into the ICC model in Sections 3.3, 3.4, and 3.5 can reduce the complexity during various engineering processes, some development phases such as testing and analysis may require additional information such as context. Context-sensitive methods greatly reduce the effort required in isolating the bugs or problems during testing and analysis. Since context information can't be included during the design phase, we excluded the information

while formalizing the model in Section 3.1. To include the context in the formal ICC model of Section 3.1, the definition of the edge needs to be extended, which now becomes: *E = <s, t, i, c> | <s, t, f, c> where i ∈I, f ∈F, c ∈C, c is a context, and C is a set of contexts in an application.*

An intent is delivered from a source to a sink component as an argument through a method call. Here, context refers to the method's calling context that is the method in which an intent delivery method has been called. A context is represented by an edge between a source and a sink component. An implicit communication has two kinds of events: an application sending intents to third-party applications and an application receiving intents from third-party applications. When an application sends intents, the method's calling context can be easily identified from the source code. However, when an application receives intents, the context can't be identified because the source code of the third-party application is not available. In this case, the method that receives the implicit intent is used as a context because the receiving method may be affected by the communication. A pending communication can have several contexts such as the context that passes the pending intent to third-party applications, the context that passes the wrapped intent from third-party applications, and the context that receives the wrapped intent. We use the method that passes the pending intent to a third-party application as a context. Including the context information in the running ICC model results into the model shown in Figure 7.

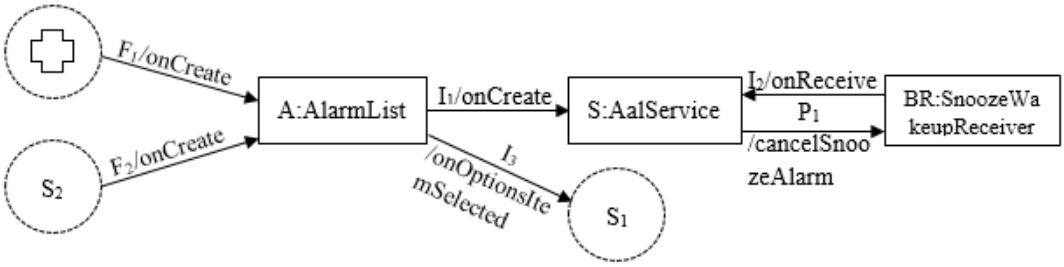

Figure 7. Modeling context for the running example.

As shown in Figure 7, the communication between Launcher and *AlarmList* and the communication between S_2 and *AlarmList* use *onCreate* as a context. These both communications are implicit in which *AlarmList* receives intents from third-party applications. According to the defined rule, the method that receives the intent is used as a context. To understand the use of *onCreate* method instead of other methods available in the *AlarmList* component, we have to understand the life cycle of the component. When an activity component is started using an intent, system calls *onCreate* method that receives the intent. Similar to the activity component, the model uses *onCreate* and *onReceive* for service and broadcast receiver components, respectively.

4. Security analysis

In this paper, security analysis is restricted to ICC. Since intra-application communication does not create any security issues, the security analysis is further restricted to inter-application communication that takes place through intents, which can create two kinds of security issues: vulnerable components can be exploited to perform certain tasks that couldn't have been allowed to perform under normal circumstances and an application can leak privacy data to malicious

applications. We present a security analysis algorithm in Figure 8 that generates warnings under both circumstances.

```
1:  Input: Manifest file M and source Code S of an application A
2:  Output: Security report R of A
3:  foreach component C declared in M
4:      if C is exported and does not declare permission
5:          addWarning(R, C is unprotected)
6:      end if
7:  end foreach
8:  foreach dynamic broadcast receiver B in S
9:      if B does not declare permission
10:         addWarning(R, B is unprotected)
11:     end if
12: end foreach
13: foreach component P in S
14:     foreach implicit intent I in P
15:         if I calls putExtra or putExtras
16:             addWarning(R, I in P may be leaking privacy data)
17:         end if
18:     end foreach
19: end foreach
```

Figure 8. Security analysis algorithm.

An Android application exposes its components for inter-application communication. In addition to benign applications, the exposed components can also be accessed by malicious applications if they are not protected, which creates security risks. Developers can protect the exposed components using permissions, which other applications must acquire before accessing the components. The mechanism restricts access to the exposed components to only known applications. However, developers may fail to protect the exposed components [28]. If an exposed component is not protected with a permission then the security analysis algorithm (line 3-7) shown in Figure 8 generates a warning. However, there is one exception to this rule. Launcher and shortcut communications are generally considered safe; therefore, the algorithm does not generate warnings for these communications. Broadcast receiver components can also be declared in the source code that needs to be protected with permissions. If an unprotected broadcast receiver is found in the source code then the algorithm (line 8-12) generates a warning.

Privacy data leak is one of the major concerns among Android application users. Data can be leaked mainly through certain API calls such as API calls for sending SMS and performing network operations and intents during inter-application communication. Here, the discussion is limited to the intent-based privacy data leaks. Various kinds of data including privacy data can be added to an intent using method calls such as *putExtra* and *putExtras*. Developers must be careful in adding sensitive privacy data to an intent, especially in the case of implicit intents. Explicit intents can be received by only known applications whereas implicit intents can be intercepted by malicious applications. The security analysis algorithm (line 13-19) shown in Figure 8 identifies implicit intents that call *putExtra* or *putExtras* method and warns developers that a component

may be leaking privacy data through a particular intent. The security report generated by the algorithm for one of the applications named *OpenSudoku* is shown in Table 1.

Table1. Security report for OpenSudoku application.

Application Name: *OpenSudoku*
Components that send intent to other apps with extra data: *None*
Intents that carry extra data to other apps: *None*
Components that receive intents from other apps and are not protected by permissions: *SudokuEditActivity, FileImportActivity, ImportSudokuActivity, SudokuImportActivity*
Intent filters used by unprotected components: *F2, F3, F4, F5*

Table 2. Comparison between security analysis algorithms of ComDroid and ICCMATT.

	ComDroid	**ICCMATT**
Target	• DEX files	• Source Code
Exported components without protection	• Component types included: activity, service, broadcast receiver, content provider • Considers a component protected if the permission used has protection level signature or signatureOrSystem.	• Component types included: activity, service, broadcast receiver • Considers a component protected if the permission used has protection level normal, dangerous, signature, or signatureOrSystem.
Intents without specific targets	• Generates a warning for each implicit intent	• ICC graph shows the intents without specific targets (implicit intents). • Does not explicitly generate warnings.
Intents with data	• Generates a warning for an implicit intent with data • Flow sensitive, intra-procedural static analysis	• Generates a warning for an implicit intent with data • Flow insensitive, intra-procedural static analysis

The security analysis algorithm presented in Figure 8 is similar to ComDroid [1]. However, there are some significant differences in approach as shown in Table 2. Our goal in this paper is not to merely reproduce the existing algorithm but to present the security information to developers in visual abstractions. To achieve the goal, we integrate key security information into the ICC model. In Section 3.2, we represented components using the annotation *<componentType: componentName>*. We extend the annotation as *<componentType: componentName>[E][P]*, where [E] and [P] represent exported and protected components respectively. Similarly, each intent is appended with data information as I_n[D], where I_n represents intent identity and [D] indicates that extra data is present in the intent I_n. Developers can analyze vulnerable components and intents

using these key information in the ICC model, for example, the ICC graph shown in Figure 11 for *OpenSudoku* application shows that none of the exported components are protected with permissions.

5. Test Case generation

Most of the existing model-based test case generation techniques in Android focus on GUI testing [35, 36, 37, 38, 39], while our technique mainly focuses on ICC. There are some existing testing techniques [2, 26, 40] that focus on ICC but they are not model-based and work on low-level abstraction. There is also another significant difference. The techniques [2, 40] define a single ICC event as a test case, whereas our technique includes multiple ICC events as a test case. The reason is very simple. Components in an application may interact in a sequence and it's important to capture the sequence of ICC events in order to test the communication behavior properly. For example, if the ICC between *AlarmList* and S_1 in Figure 7 is tested in isolation then it may not produce a correct result because predecessor ICC between S_2 and *AlarmList* that may affect all of its successor ICC has not been taken into account.

5.1 Test case generation algorithm

The proposed ICC model contains nodes and edges. Launcher, shortcuts, components, and pseudo-components (entry points for implicit intents and exit points for implicit intents) are represented by nodes. Intents, intent filters, and context are represented by edges. The aim is to obtain a sequence of end-to-end ICC events that means from one of the entry points to one of the exit points of the model. Each such sequence of ICC events represents a test case. Entry points and exit points are represented by entry nodes and exit nodes respectively. Entry nodes are those nodes through which a user can enter into the application. Launcher, shortcuts and entry node for implicit communication are the entry nodes in the model. Exit nodes are those nodes through which a user can jump to a third-party application or no further communication through an intent is allowed. However, an exit node can communicate to itself through an intent. Exit nodes for implicit communication and the components that do not send an intent to other components are the exit nodes in the model.

Here, we further clarify entry nodes and exit nodes in terms of an ICC graph and then define pre-conditions and a post condition for generating test cases from the ICC graph. *An ICC graph G is represented with a 2-tuple (N, E), from Section 3.1. Entry nodes in the ICC graph G are EN, where* $EN \in N$. *For each entry node i in EN,* $(N\text{-}EN_i) \rightarrow EN_i$ *does not exist in G. Exit nodes in the ICC graph G are EX, where* $EX \in N$. *For each exit node j in EX,* $EX_j \rightarrow (N\text{-}EX_j)$ *does not exist in G. The test cases can be generated only if the ICC graph G satisfies the pre-conditions: all nodes N in G must be directly or indirectly reachable from one of the entry nodes in EN and all nodes N, except exit nodes EX, in G must be directly or indirectly reachable to one of the exit nodes in EX. The test case generation terminates when it satisfies the post-condition: all the generated test cases combined must contain all edges in E at least once.* If we take the ICC graph in Figure 7 as an example then we have *LAUNCHER* and S_2 as entry nodes and S_1 as an exit node. Test cases can't be generated from the ICC graph because it fails to satisfy the pre-condition. Nodes *AalService*

and *SnoozeWakeupReceiver* can't reach to an exit node (S_1) from any of the entry nodes (*LAUNCHER* and S_2). However, *AalService* has other outgoing edges in the real application.

The ICC graph generated through the proposed modeling technique has several properties. First, the graph may be disconnected. Second, the graph may be a multigraph with parallel edges. Third, the graph may contain several entry and exit nodes. Fourth, the graph may be cyclic. And fifth, the graph may contain loops (pseudograph). Considering all these properties, we have proposed a test case generation algorithm shown in Figure 9. The test case generation algorithm uses both systematic and random approaches. The combined approach is better than systematic or random approaches [41] because systematic approaches may not mimic the real scenarios and random approaches may not cover enough scenarios.

As shown in Figure 9, the test case generation algorithm takes an ICC graph as input and generates a file containing test cases as output. The algorithm terminates when all the edges of the graph are visited at least once (line 7). Since we intend to take end-to-end ICC events as a single test case, the algorithm first picks up an entry node and starts the test case (line 8, 12 and 13). The test case terminates when it reaches to one of the exit nodes (line 14). The algorithm traverses from an entry node to an exit node by choosing an edge at a time randomly with some guidance (line 14-44). The algorithm first checks for all the unvisited outgoing edges of a node. If it finds an unvisited edge then it selects the edge (line 17-21) and if it does not find an unvisited edge then it checks for the edges that have not been visited in the current test case. If it also does not find an unvisited edge in the current test case then it selects an edge randomly from all the visited edges (line 23-24). However, if it finds unvisited edges in the current test case then it selects an unvisited edge randomly (line 26).

Once an edge has been selected, the algorithm adds the edge and the target node of the edge in the current test case (line 36-37). The algorithm adds the selected edge in the test case to differentiate among the parallel edges. If the selected edge has not been previously visited then it is marked as a visited edge (line 39-41). Now, the target node of the selected edge acts as the new source node (line 38). If the source node is one of the exit nodes then the test case terminates otherwise the process of selecting an outgoing edge repeats. Once the test case terminates, the algorithm checks it for an unvisited edge. If it finds an unvisited edge in the test case then the algorithm writes the test case to a file (line 45-47). Since the algorithm uses a random technique, there are chances that the test case may not contain an unvisited edge. The test case that does not contain an unvisited edge is simply discarded.

```
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 <- an element of entryNodes
13:         add(testCase, sourceNode)
14:         while exitNodes does not contain sourceNode
15:             Declare ArrayList: outEdges; Boolean: exitFound <- false
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, sourceNode)
33:                     exitFound <- true
34:                 end if
35:                 if (! exitFound)
36:                     append(testCase, chosenEdge)
37:                     append(testCase, target node of chosenEdge)
38:                     sourceNode <- target node of chosenEdge
39:                     if chosenEdge is not in visitedEdgeList
40:                         add(visitedEdgeList, chosenEdge)
41:                     end if
42:                 end if
43:             end if
44:         end while
45:         if testCase contains at least one unvisited edge
46:             add(F, testCase)
47:         end if
48:     end foreach
49: end while
```

Figure 9. Test case generation algorithm.

The algorithm for selecting exit nodes (line 5) fails when a node has a single outgoing edge to itself in an ICC graph. If the node is selected as an exit node at the start of the test case generation algorithm then the outgoing edge to itself will not be covered in the test case due to which the algorithm uses a dynamic approach. If the immediate predecessor edge visited in the test case is same as the selected edge and the source node has a single outgoing edge then the algorithm (line 30-34) selects the source node as an exit node and terminates the test case.

The test cases generated by the algorithm is not executable in its original form; therefore, we modify the generated test cases to make them executable. For example, one of the test cases generated by the algorithm from the ICC graph shown in Figure 11 of *OpenSudoku* application is *LAUNCHER→F_1/onCreate→FolderListActivity→I_8/onListItemClick→SudokuListActivity→I_{17}/playSudoku→SudokuPlayActivity→I_{10}/onOptionsItemSelected→GameSettingsActivity*. The test case can't be executed in its current form because it lacks two vital elements. It does not model the life cycle events of the components and it does not include method call sequence from the start of the component to the context where an intent is delivered to another component. For example, in the test case, *SudokuListActivity* starts *SudokuPlayActivity* through an intent I_{17} by calling *playSudoku* method but the test case does not indicate method call sequence for the *playSudoku* method in *SudokuListActivity*. Table 3 shows the steps performed to make the test case executable and the resulting test case in each step.

Table 3. Steps for making test cases executable.

Steps	Resulting Test Case
ICC graph traversal	*LAUNCHER→F_1/onCreate→FolderListActivity→I_8/onListItemClick→SudokuListActivity→I_{17}/playSudoku→SudokuPlayActivity→I_{10}/onOptionsItemSelected→GameSettingsActivity*
Incorporating life cycle events	*LAUNCHER→F_1→FolderListActivity→onCreate→onStart→onResume→onListItemClick/I_8→SudokuListActivity→onCreate→onStart→onResume→playSudoku/I_{17}→SudokuPlayActivity→onCreate→onStart→onResume→onOptionsItemSelected/I_{10}→GameSettingsActivity*
Removing unnecessary life cycle events	*LAUNCHER→F_1→FolderListActivity→onCreate→onStart→onListItemClick/I_8→SudokuListActivity→onCreate→onResume→playSudoku/I_{17}→SudokuPlayActivity→onCreate→onResume→onOptionsItemSelected/I_{10}→GameSettingsActivity*
Incorporating method call sequence	*LAUNCHER→F_1→FolderListActivity→onCreate→onStart→onListItemClick/I_8→SudokuListActivity→onCreate→onResume→onContextItemSelected→playSudoku/I_{17}→SudokuPlayActivity→onCreate→onResume→onOptionsItemSelected/I_{10}→GameSettingsActivity*

We first modify the generated test cases to incorporate life cycle events. A component's life cycle is implemented through callback methods. The execution sequence of the callback methods depends on the component type and how the component is started or stopped. Table 4 shows the execution sequence of callback methods for each component type, for example, if an activity component is started using *startActivity* then the component's *onCreate*, *onStart*, and *onResume* callback methods are executed in sequence. The example of resulting test case after incorporating life cycle events is shown in Table 3. This step naively inserts callback methods depending on

component types and intent delivery methods. However, a component is not required to implement all the callback methods. Therefore, the next step removes the callback methods that have not been implemented by the component, resulting in test cases with the proper order of life cycle events.

Table 4. Execution sequence of callback methods for different types of components.

Target Component Type	**Intent Delivery Methods**	**Execution Sequence of Callback Methods**
Activity	startActivity, startActivityForResult, getActivity, getActivities	onCreate, onStart, onResume
Activity	setResult	onRestart, onStart, onResume, onActivityResult
Service	startService, getService	onCreate, onStartCommand
Service	bindService	onCreate, onBind
Service	stopService	onDestroy
Broadcast	sendBroadcast, getBroadcast	onReceive

Finally, we generate method call sequence for each method (context of ICC graph) present in the test case. The returned call sequence is appended to the test case. If there are multiple paths of call sequence then only one call sequence path is selected because our goal is to execute the method that triggers ICC. For the example test case shown in Table 3, we generate method call sequence for *onListItemClick*, *playSudoku*, and *onOptionsItemSelected* methods. The methods *onListItemClick* and *onOptionsItemSelected* are not called by any other methods therefore, they are not modified in the test case. However, the method *playSudoku* is called by *onContextItemSelected* and *onListItemClick* methods that in turn are not called by any other methods. Here, we have two different paths for executing *playSudoku* method but we take only one path and modify the test case as shown in Table 3.

The resulting test cases are executable and can identify various reliability issues including issues caused by improper state management, which is one of the major cause of bugs in Android applications [9]. However, we do not provide a test execution technique in this paper. There are further steps required to automatically execute the resulting test cases, particularly input generation and test automation, which we intend to do in future. Existing techniques [42, 38] can execute our test cases but the tools are not available for public use. AppIntent [42] is the best-suited technique that performs guided symbolic execution to generate inputs and then automates the test execution using InstrumentationTestRunner [43].

5.2 Test case coverage criterion

It is important to evaluate the adequacy of the obtained test suite. The test suite obtained through the proposed model-based test case generation technique should adequately address the critical section of the target domain. Coverage criteria are most-widely used for evaluating the adequacy of a test suite. In this paper, we propose an intent and intent filter (I-IF) coverage criterion that can be used to measure the adequacy of the test suite obtained from the proposed model. In I-IF coverage criterion, each intent and intent filter should be covered at least once. The percentage of I-IF coverage is calculated by dividing the total number of intents and intent filters covered by

the model with the total number of intents and intent filters associated with all the components of an application. The resulting value is multiplied by 100.

$$\text{I} - \text{IF coverage (\%)} = \frac{\text{Total number of intents and intent filters covered by the model}}{\text{Total number of intents and intent filters in components of an application}} \times 100$$

The I-IF coverage is basically a branch coverage criterion because intents and intent filters are represented by edges in the model. Components in the above equation represent all the component types except content provider. Remember, the test case generation algorithm terminates only when all the edges are covered that does not mean it provides 100% coverage because the ICC graph may not have modeled all the ICC events of an application. The I-IF coverage represents the percentage of intents and intent filters covered from source code. There are several reasons for selecting the I-IF coverage criterion. An intent is an important entity because it may carry unexpected data from an entry node to other nodes that can cause unexpected behavior. An intent filter acts as a constraint against receiving implicit intents; therefore, it is important to stop unexpected data flowing inwards at the first place. There are other important entities in the model such as entry nodes and exit nodes for implicit intents. Entry nodes are the points through which an application can get unexpected data causing unexpected behavior. Since the test case generation algorithm covers all the intents and intent filters present in the model, the entry and exit nodes will be automatically covered.

6. ICCMATT implementation

Conceptual models are primarily built during design phase from the specification document. Its main purpose is to directly or indirectly assist stakeholders in completing the development task. Due to the current development practices of Android applications, it is very difficult to obtain the requirement specification document; therefore, we have developed a tool named ICCMATT that extracts an ICC graph from the source code instead of building a tool to model ICC from the specification document. Moreover, the tool also generates test cases and a security report. The primary goal of the tool is to assist developers in analyzing ICC during application development.

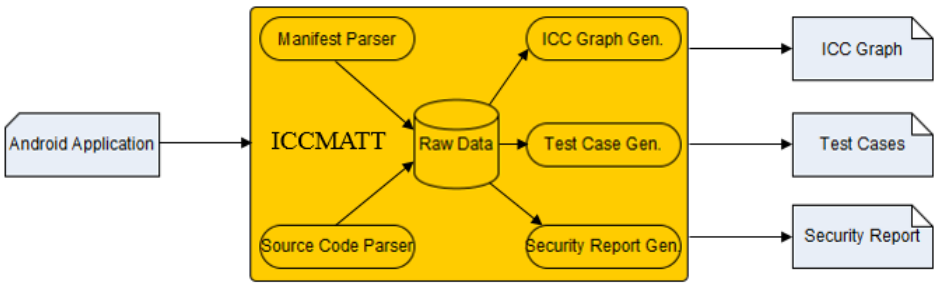

Figure 10. High-level design of ICCMATT.

ICCMATT is an open source Eclipse plug-in tool [15] written in Java. The high-level design of ICCMATT is shown in Figure 10. It takes an application's source code as input and generates three different files as outputs: a text file containing all the test cases, another text file containing a security report, and a GraphML [44] file containing an ICC graph. Since the target users of the tool are application developers, we chose to work at source code level. ICCMATT consists of five

major modules: manifest parser, source code parser, ICC graph generator, test case generator, and security report generator. Manifest parser and source code parser modules extract required data from the application that are later used by other modules. An ICC event has a source node, a target node, an intent or intent filter, and a context. All these entities forming a single ICC event are identified and stored in a structured format that later simplifies the tasks of generating an ICC graph and test cases through the ICC graph generator and the test case generator modules respectively. A security report is also generated from the extracted data through the security report generator module. Some of the data required for security report are stored along with entities of ICC, while some data are stored separately.

ICCMATT takes project name as input and parses the manifest file using DOM parser. It collects name and type of all the components declared in the manifest file, which is later used to identify the components. The component names are also utilized later by the source code parser module to parse only those class files that match to the component names, which reduces the execution time of the tool. The tool also checks for protection against the components while parsing the manifest file. It checks whether a component is protected by a permission and stores the data, which is later utilized by the security report generation module for security analysis. The final task of the manifest parser module is to collect and store data of intent filters. For each intent filter declared against a component inside the manifest file, it stores a source, a target, an intent filter, and a context. A pseudo-component with a unique identity is stored as a source. The unique identity is generated by the tool. The component against which the intent filter has been declared is stored as a target. The tool generates and stores a unique identity for each intent filter. A context is decided based on the type of the component against which the intent filter has been declared. The tool assigns *onCreate*, *onCreate*, and *onReceive* as a context for activity, service, and broadcast receiver components respectively. In an application, a launcher and shortcuts are declared through intent filters. If the manifest parser finds such intent filters then it stores LAUNCHER or SHORTCUT depending upon the type as source name of the pseudo-component, instead of a unique identity.

Source code parser module that extracts most of the data required by other modules is the main module of ICCMAAT. It uses AST Parser [45] to extract the data from the source code. The component names extracted by the manifest parser module are used to identify and parse only those files that represent components. The files representing the components are fed into the source code parser module one at a time. The module takes the source code inside a method as a unit of processing. Rather than parsing all the source code inside a file and then processing it, the module parses and processes the code inside a method and then proceeds to another method within the file.

ICC takes place through intents that contain various information such as the name of the source and target components that are required to generate an ICC graph. The source code parser module checks for the intent instantiation code inside a method. It also checks for the methods declaring an intent as a return type because an intent can be instantiated in other than a component class and then passed to a component class through method calls. In such cases, the module finds the class where the intent has been originally instantiated by generating method call sequence and then process the intent instantiation code. An intent can be instantiated in six different ways using six

different public constructors as shown in Table 5. The module checks all types of constructors used for instantiating an intent except the copy constructor defined in number 2. If it finds an intent instantiation code then it differentiates between an explicit and an implicit intent using the type of constructor. The constructor number 5 and 6 explicitly declare a target component in the form of a component class; therefore, the module can easily identify the intents as explicit intents. The remaining constructors do not explicitly declare a target component; therefore, the intents are identified as implicit intents if and only if the intents do not set a target component by invoking one of the public methods listed in the last column of Table 5. Due to technical limitations, currently, the module can identify and process only *setClass* and *setClassName* methods, which are the most frequently used methods. If the module finds *setComponent* or *setPackage* being invoked by an intent, it ignores and places that intent into "*not covered*" list.

Table 5. Public constructors and methods for an intent.

Number	**Public constructors for intent**	**Public methods**
1	Intent()	setClass (Context packageContext, Class<?> cls)
2	Intent(Intent o)	setClassName (Context packageContext, String className)
3	Intent(String action)	setClassName (String packageName, String className)
4	Intent(String action, Uri uri)	setComponent (ComponentName component)
5	Intent(Context packageContext, Class<?> cls)	setPackage (String packageName)
6	Intent(String action, Uri uri, Context packageContext, Class<?> cls)	

For each implicit intent, the module stores a source and a target component. The class representing the component in which the implicit intent has been instantiated is stored as a source component, whereas a pseudo-component with a unique identity is stored as a target component. Similar to an implicit intent, the module stores a source and a target component for each explicit intent. However, finding a source component of an explicit intent is not straight forward. One widely used practice in Android applications development is the use of context object [46], which is used for starting components, among various other purposes. The public constructors or methods of an intent use the context object representing an activity or a service component as a source component. When a class name of a component or *this* keyword representing the current component is directly passed as a context object, the module stores the name of the component as a source component. Meanwhile, a context object can also be created in another class and then passed to the constructors or methods. In such a case, the module resolves the context object by finding the component representing the context object by constructing a call graph. If the module could not resolve the context object then it completely ignores the intent and places the intent into "*not covered*" list. Moreover, an application can also be used as a context object. In such a case, the class representing the component in which the explicit intent has been instantiated is stored as a source component. In the case of an explicit intent, a target component is explicitly set through

public constructors or public methods. The module checks the last argument of the constructors or methods for a class name representing a component and stores that as a target component.

The source code parser module checks for the intent instantiation code and finds a source and a target component as described above. If it correctly identifies a source and a target component then it checks for the methods that can deliver the intent from the source to the target component. If it finds any one of the intent delivery methods listed in Table 4 and the intent that has been instantiated is same as the intent that has been used in the method then it assigns and stores a unique identity for the intent. In the case of intent delivery methods *getActivity*, *getActivities*, *getService*, and *getBroadcast*, the intent is identified and stored as a pending intent. An intent can be used for other than ICC; therefore, the module may not find any intent delivery methods listed in Table 4. In such a case, the module removes the stored source and target components of the intent and places the intent into "*not covered*" list.

The source code parser module needs to collect one more entity called context to complete an ICC graph. The module parses the source code of a method as a unit. If the module finds an intent instantiation code in a method then the method is used as a context. An intent can also be instantiated in a method and returned to another method where the intent is delivered to a target component. Here, the method from where the intent is delivered to a target component is used as a context. Along with all the aforementioned tasks, the source code parser module performs the task of identifying dynamic broadcast receivers. It checks for the code that invokes the *registerReceiver* method. If it finds the code then it stores a pseudo-component with a unique identity as a source component, a broadcast receiver component with a unique name as a target component, an intent filter with a unique identity, and *onReceive* method as a context. Although the module has collected all the data required for an ICC graph and test cases generation, it needs to collect more data for security analysis. Before collecting a context, the module checks whether the intent invokes *putExtra* or *putExtras* method. The module stores the information, which is later utilized for security report generation.

Data extracted by the manifest parser and the source code parser modules are used by the ICC graph generator, the test case generator, and the security report generator modules to generate an ICC graph, test cases, and a security report respectively. The task of generating an ICC graph is trivial because all the necessary data have already been acquired. The ICC graph generator module populates nodes of the ICC graph from the data of source and target components. All the unique components from both the source and target components are taken as nodes of the graph. Edges are then populated between two nodes representing a source and a target component by taking an intent or an intent filter and a context from the extracted data. The module uses the GraphML writer library blueprints [47] to store the ICC graph in GraphML format. Reasons behind using the GraphML format are its scalability and flexibility in analyzing, manipulating, and maintaining a graph. The test case generator module generates test cases by using the algorithm and the technique described in Section 5.1. The security report generator module generates a security report from the extracted data using the algorithm described in Section 4.

7. Evaluation of ICCMATT

ICCMATT is a completely automated Eclipse plug-in tool. The main purpose of the tool is to assist developers in identifying improper handling of ICC during application development. There are no hard rules for identifying improper handling of ICC that may include common developer mistakes. For example, a developer may perform an implicit communication instead of an explicit communication, which invites security vulnerabilities. Similarly, a developer may leave a component exposed to third-party applications [28]. While advised against it in general, a developer may deliberately perform the task. Thus, only a developer who has developed the application can make the correct decision about improper ICC. The ICCMATT tool facilitates developers in making the decisions effectively and efficiently.

Table 6. Benchmark applications.

No.	App Name	Size	No.	App Name	Size	No.	App Name	Size
1	ADBM	10	**31**	k9mail	39	**61**	PSIAndroid	3
2	androidDreamCPU	8	**32**	Kaleidoscope	4	**62**	Rainwave	5
3	AppAlarm	17	**33**	KeePassDroid	16	**63**	Ray diagrams	6
4	AppsOrganizer	14	**34**	KindMind	12	**64**	reddit is fun	15
5	AsciiCam	7	**35**	L9Droid	10	**65**	reminders_master	10
6	Auto-Away	7	**36**	ListMyApps	4	**66**	RemoteKeyboard	7
7	Avare	19	**37**	LoginActivity	17	**67**	RingdroidSelect	3
8	BarcodeBox	4	**38**	MainMenuActivity	6	**68**	Roaming Info	4
9	BatteryFu	8	**39**	MainPreferences	11	**69**	rtl_tcpAndroid	3
10	Bitcoin Paper	4	**40**	MinistocksActivity	12	**70**	SandwichRoulette	7
11	CameraTimer	5	**41**	MultiPictureLiveWall	13	**71**	sanity	41
12	campyre	9	**42**	Muspy for Android	16	**72**	Scribbler	4
13	ChannelListActivit	10	**43**	NDKmolActivity	6	**73**	SearchDataActivity	6
14	connectbot	12	**44**	NewNote	9	**74**	SelectUserActivity	6
15	DefCol	6	**45**	NoteListActivity	5	**75**	Silent Night	6
16	DeskCon	14	**46**	NotificationStopwatch	4	**76**	Simple c25k	6
17	diasporawebclient	6	**47**	NSTools	4	**77**	SimpleDo	6
18	droidparts-battery	4	**48**	NWSWeatherAlerts	9	**78**	SplashActivity	14
19	Email Popup	5	**49**	OpenFixMap	5	**79**	STK Addon Viewer	7
20	EnvelopesActivity	4	**50**	OpenSudoku	10	**80**	SwiFTP Lib	8
21	EZ Wifi Notifica.	4	**51**	PasteeDroid	3	**81**	TimerDroid	7
22	falling for reddit	18	**52**	PathfinderOpenRef	5	**82**	TintBrowser	4
23	FeederActivity	21	**53**	PenroserActivity	6	**83**	TitleScreen	8
24	file manager	8	**54**	Periodical	5	**84**	ToDoWidget	20
25	GaAT	6	**55**	PeriodicTableActivity	5	**85**	tomdroid	8
26	GeneratorActivity	3	**56**	Plakate	4	**86**	TwistedHomeManage	5
27	gnupg-for-android	24	**57**	PluckLock	4	**87**	UPM-Android	14
28	GoogleApps	3	**58**	PlusMinusTimesDivid	6	**88**	Vanilla Music	17
29	Ham	6	**59**	portal-timer	3	**89**	VoiceNotify	4
30	HungarianRings	3	**60**	PrettyGoodMusicPlaye	7	**90**	VPlug	3

The tool provides vital information to developers in the form of an ICC graph and a security report. It is up to the developers to make correct decisions. However, the generated test cases can be executed to identify reliability issues caused by incorrect implementations of ICC. In this section, we evaluate efficiency and effectiveness of ICCMATT in generating an ICC graph, a

security report, and test cases. We also evaluate the adequacy of the generated test cases through the I-IF coverage criterion.

We evaluated ICCMATT tool on 90 active and open source Android applications listed in Table 6. The size column in Table 6 represents number of components declared by the application in the manifest file. Size in terms of number of components is more relevant than the number of lines of code for ICC evaluation. We selected all the benchmark applications from F-Droid [48], which is a repository of free and open source Android applications. We browsed F-Droid during June 2015 and downloaded all those applications that were hosted on GitHub. This criterion gave us 473 applications from which we removed those applications that were either not active or contained less than three components. The remaining applications were imported in Eclipse. Some of the applications could not be imported due to errors. Finally, we removed those applications that had less than three intents, which gave us the final benchmark size of 90 applications. The evaluation was performed on Windows 7 OS with Intel Core i5 processor on 8 GB RAM. Eclipse Juno was used as an IDE.

7.1 Evaluation of ICC graphs

One of the outputs generated by ICCMATT is an ICC graph in GraphML format. We evaluated completeness and correctness of the generated ICC graph for each benchmark application. The evaluation results are shown in Table 7. Completeness represents the percentage of ICC events extracted in the ICC graph from the source code. Correctness represents the percentage of correct ICC events in the ICC graph.

In most of the cases, as indicated by the completeness column in Table 7, ICCMATT effectively extracted an ICC graph from the source code. The reasons behind extraction of incomplete ICC graphs are due to the limitations of ICCMATT. Being a static analysis tool, it can't resolve intents dynamically, for example, arguments passed in an intent object at runtime or an intent object instantiated by the ternary expression *condition ? new intent : new intent*. The tool also can't resolve *setComponent* or *setPackage* used for explicitly setting target component names as described in Section 6. Finally, if the tool fails to resolve a context or fails to match an intent at intent instantiation point and in the intent delivery method then the tool does not extract the ICC event. Since the tool discards the ICC events that it can't resolve properly, most of the extracted ICC graphs are accurate, as indicated by the correctness column of Table 7. Inaccuracies in the extracted ICC graphs are only caused by false positive of extra data present in the intent. In all the inaccurate cases, extra data is added to an intent depending on some conditions, which the ICCMATT tool fails to detect.

The ICC graph generated by ICCMATT for *OpenSudoku* application is shown in Figure 11. Currently, the tool cannot directly open GraphML file to view the ICC graph. There are several tools available in the market that can be used for this purpose such as Gephi [49] and yEd [50]. Gephi is an open source tool but it can't currently display parallel edges. yEd is not an open source tool but it is freely available and it has all the properties required to view the ICC graph. We used yEd to view the graph. There are some minor shortcomings in the ICCMATT. It does not generate the shape defined for Launcher and Shortcut; therefore, it uses names LAUNCHER and SHORTCUT respectively. It generates all the nodes in a single shape. We manually changed the

shape of the components to rectangle in Figure 11. It also generates solid lines for all the nodes. We changed that too manually from solid to dotted lines in case of pseudo-components.

Table 7. ICC graph evaluation.

App No.	Completeness	Correctness	App No.	Completeness	Correctness	App No.	Completeness	Correctness
1	100	100	**31**	85	100	**61**	100	100
2	100	100	**32**	100	100	**62**	85	100
3	98	97	**33**	91	100	**63**	14	100
4	81	100	**34**	92	100	**64**	94	100
5	100	100	**35**	100	94	**65**	72	100
6	100	95	**36**	100	100	**66**	100	100
7	76	100	**37**	93	100	**67**	89	100
8	100	93	**38**	100	100	**68**	100	100
9	100	100	**39**	78	96	**69**	89	100
10	75	100	**40**	100	96	**70**	63	100
11	100	100	**41**	96	100	**71**	98	100
12	69	100	**42**	89	100	**72**	83	100
13	100	100	**43**	82	100	**73**	100	100
14	100	100	**44**	100	100	**74**	100	100
15	100	100	**45**	100	100	**75**	100	100
16	89	100	**46**	100	100	**76**	100	100
17	100	100	**47**	78	100	**77**	100	100
18	83	100	**48**	95	100	**78**	93	100
19	91	100	**49**	100	100	**79**	100	86
20	67	100	**50**	100	91	**80**	95	100
21	100	100	**51**	100	100	**81**	100	100
22	75	100	**52**	100	100	**82**	100	100
23	90	100	**53**	76	100	**83**	92	100
24	92	100	**54**	100	100	**84**	80	100
25	80	100	**55**	100	100	**85**	100	100
26	40	100	**56**	75	100	**86**	81	100
27	100	100	**57**	100	100	**87**	100	92
28	100	100	**58**	50	100	**88**	63	100
29	95	100	**59**	83	100	**89**	69	100
30	100	100	**60**	100	98	**90**	83	100

The ICC graph in Figure 11 provides a complete picture of ICC in *OpenSudoku* application. A developer can find various information about the application in the graph. For example, the application has ten activity components and *FolderListActivity* is the main component. The application performs intra-application communication among its components through explicit intents. The application also interacts with third-party applications. Four of its components are exported and can receive intents from third-party applications S_1, S_2, S_3, and S_4. None of the exported components are protected with permissions. The *FolderListActivity* component communicates with a third-party application S_5 without sending any sensitive data. All these information at one place immensely help developers in identifying improper ICC.

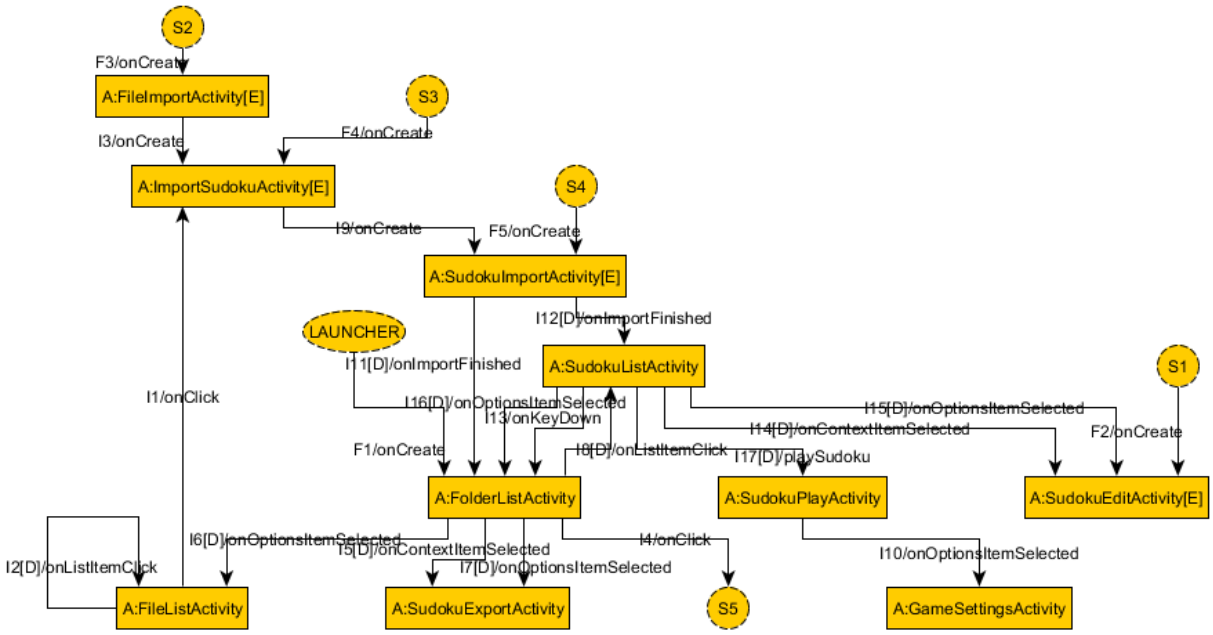

Figure 11. ICC graph generated by ICCMATT for OpenSudoku application.

7.2 Evaluation of security reports

Another output generated by ICCMATT is a security report. The security report for *OpenSudoku* application is shown in Table 1. Similar to ICC graphs, we evaluated completeness and correctness of the generated security report for each benchmark application. The tool completely and correctly generated security reports for all the applications except one. The tool incorrectly identified an intent leaving application boundary with data in *BarcodeBox* application, whereas the intent does not contain any extra data. The reason behind the false positive is described in Section 7.1.

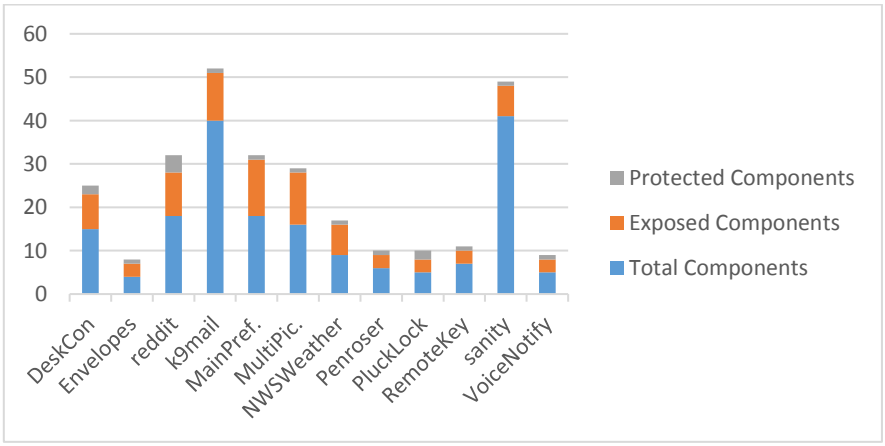

Figure 12. Applications with exported but partially protected components.

A developer can find vulnerable components and intents by analyzing security information displayed in the ICC graph. The security report generated by the ICCMATT works as a supplementary data for security analysis. As shown in Table 1, the security report generated by

ICCMATT for *OpenSudoku* application indicates that none of the exposed components are protected by permissions. It also indicates that the intent I_4 does not contain any data. Out of 90 benchmark applications, security reports generated by ICCMATT indicate that 25 applications do not expose any of its components to third-party applications. In the remaining 65 applications that expose one or more components, only 12 applications protect their components by permissions. Even in those 12 applications, number of components protected are far less than the number of exposed components as shown in Figure 12. The figure includes dynamic broadcast receiver components. The security reports also indicate that 44 applications do not send any data to third-party applications. The remaining 46 applications send data to third-party applications through one or more intents, which does not mean these applications leak privacy data. Finding privacy data leak would require more precise data analysis but the tool currently does not support such analysis.

7.3 Evaluation of test case generation

Third and the final output generated by ICCMATT is a file containing test cases. Evaluating the correctness of the generated test cases does not require manual intervention. The test case generation algorithm terminates only when all the edges of the ICC graph are covered. The algorithm does not execute if the generated ICC graph does not satisfy the pre-conditions. Out of 90 benchmark applications, the tool could not generate test cases for eight applications as shown in Table 8. In *ADBM*, *KeePassDroid*, *AppsOrganizer*, and *PenroserActivity* applications, some of the nodes in the generated ICC graph could not be reachable from one of the entry nodes, whereas in *DefCol*, *FeederActivity*, *NewNote*, and *sanity* applications, any of the exit nodes were not reachable from some of the nodes of the generated ICC graph.

We measured the adequacy of the generated test cases for each benchmark application through I-IF coverage criterion. As shown in Table 8, most of the applications have I-IF coverage more than 70%. The major reasons behind missing intents by source code parser module of ICCMATT are described in Section 7.1. Along with those reasons, the tool deliberately discards the intents that do not represent an ICC event. In most of the discarded cases, intents were passed as arguments in *putExtra* or other methods. In some cases such as *FeederActivity*, *Muspy*, and *MainMenuActivity*, the tool found intents that were instantiated but not used.

Along with the coverage, we also measured number of test cases generated (NT), test cases generation time (TGT) in seconds, and tool execution time (TET) in seconds. The tool execution time represents the time taken by the tool to generate an ICC graph, a security report, and test cases. The results are shown in Table 8. In most of the cases, the test case generation took less than one second whereas the tool execution took around 1 or 2 seconds because most of the Android applications are small sized as shown in Table 6. For complex applications like *K9mail*, it took around one minute for test case generation and around two minutes for tool execution. The results indicate that ICCMATT, being a static analysis tool, has reasonable execution time. Since the test case generation algorithm uses a random technique, the number of generated test cases, the test generation time, and the tool execution time may vary. Overall, the evaluation results indicate that ICCMATT can perform its tasks effectively and efficiently.

Table 8. Test case and ICCMATT tool evaluation for benchmark applications.

App No.	NT	I-IF	TGT	TET	App No.	NT	I-IF	TGT	TET	App No.	NT	I-IF	TGT	TET
1	-	-	-	0.8	**31**	59	78	46.1	108.0	**61**	2	100	0.6	1.4
2	7	100	0.4	2.3	**32**	3	100	0.3	1.0	**62**	6	85	0.7	1.4
3	18	94	4.4	6.6	**33**	-	-	-	2.3	**63**	1	14	0.003	0.4
4	-	-	-	1.5	**34**	7	79	0.2	1.5	**64**	16	94	6.0	9.4
5	8	100	0.6	1.8	**35**	6	100	0.6	1.3	**65**	39	67	2.3	5.2
6	8	100	2.0	2.9	**36**	6	100	0.4	1.0	**66**	6	100	0.3	1.1
7	24	76	0.3	2.5	**37**	14	81	2.1	3.9	**67**	3	89	1.0	1.7
8	12	94	0.7	1.5	**38**	5	90	0.4	1.7	**68**	3	100	0.03	0.5
9	12	100	0.5	3.8	**39**	17	72	2.7	5.9	**69**	6	89	0.3	1.0
10	5	67	1.3	2.3	**40**	22	100	0.7	1.4	**70**	4	63	0.2	0.8
11	5	100	0.3	1.5	**41**	18	81	0.3	1.2	**71**	-	-	-	1.5
12	7	69	0.6	1.4	**42**	10	48	0.3	2.1	**72**	4	83	0.05	0.5
13	16	100	2.3	4.0	**43**	6	82	0.5	1.2	**73**	3	100	0.2	0.9
14	16	100	2.1	4.4	**44**	-	-	-	0.6	**74**	2	100	0.06	0.8
15	-	-	-	0.7	**45**	6	100	0.5	1.2	**75**	5	100	0.3	1.0
16	20	89	0.5	4.3	**46**	7	92	0.4	0.9	**76**	9	100	0.7	1.2
17	8	100	1.0	1.9	**47**	3	78	0.5	1.5	**77**	2	100	0.4	1.2
18	9	78	0.4	1.0	**48**	17	87	0.3	1.3	**78**	13	93	1.6	2.9
19	7	91	0.4	1.4	**49**	3	100	0.3	0.8	**79**	4	78	0.3	0.8
20	4	67	0.3	1.9	**50**	8	85	0.9	1.8	**80**	15	95	0.3	1.6
21	7	100	1.3	2.2	**51**	4	100	0.3	0.8	**81**	4	100	0.3	1.2
22	13	75	0.1	1.8	**52**	8	100	0.9	1.6	**82**	6	80	0.3	1.0
23	-	-	-	4.9	**53**	-	-	-	0.7	**83**	3	92	0.7	1.2
24	13	92	1.0	1.8	**54**	4	100	0.4	1.1	**84**	18	80	0.1	3.2
25	4	71	0.04	1.1	**55**	2	100	0.4	0.9	**85**	12	88	4.1	5.2
26	1	40	0.2	0.8	**56**	2	75	0.1	0.7	**86**	11	81	0.3	0.9
27	35	95	2.7	4.2	**57**	5	100	0.1	0.6	**87**	16	100	1.0	1.8
28	3	100	0.2	0.7	**58**	2	50	0.1	0.9	**88**	17	62	9.5	17.0
29	13	86	0.8	1.6	**59**	2	83	1.5	3.0	**89**	10	69	0.3	1.5
30	3	100	0.3	0.8	**60**	30	92	1.4	3.3	**90**	3	83	0.01	0.5

7.4 Threats to validity

ICCMATT has some limitations that may affect the results. The limitations are both general and specific in nature. Being a static analysis tool, it can't handle dynamic events, for example, a target component name passed to an intent object during runtime. Another general problem that the tool can't handle is java reflection. Researchers have proposed tools and techniques to handle reflection in Android [51, 52]. Most of the specific limitations of the ICCMATT have been discussed in Sections 7.1 and 7.3. One major limitation of ICCMATT is that it performs flow insensitive analysis, which means the tool does not take into account the conditional statements.

8. Conclusion and future works

In this paper, we have proposed a conceptual model to represent the ICC in Android applications. The proposed model can be designed from the specification document or extracted from the source code. We have developed and presented a completely automated tool to extract

the model from the source code. Moreover, the tool generates test cases and a security report based on the respective algorithms presented in this paper. With the goal of developing high-quality secure applications, the tool assists developers in identifying improper ICC. The tool generates an ICC graph that presents the entire ICC of an application in a single place, which reduces the complexity in analyzing and identifying improper ICC. The security report generated by the tool provides supplementary but precise information about the possible security loopholes. The generated test cases can be executed to identify reliability issues caused by incorrect ICC implementations.

Currently, the tool does not provide support for executing the generated test cases. There are existing techniques based on symbolic execution [42, 38] that can execute the generated test cases. However, the techniques are inefficient; therefore, we intend to develop an efficient test case execution technique in future.

References

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

[2] Maji AK, Arshad FA, Bagchi S, Rellermeyer JS. An empirical study of the robustness of inter-component communication in Android. *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* 2012; 1-12.

[3] Li L, Bartel A, Bissyande TFDA, Klein J, Traon YL, Arzt S, Rasthofer S, Bodden E, Octeau D, and McDaniel P. IccTA: detecting inter-component privacy leaks in android apps. *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE)* 2015.

[4] Ahmad W, Kästner C, Sunshine J, and Aldrich J. Inter-app communication in Android: developer challenges. *Proceedings of the 13th Working Conference on Mining Software Repositories* 2016; 177-188.

[5] Jha AK and Lee WJ. An empirical study of collaborative model and its security risk in Android. *Journal of Systems and Software* (2017).

[6] Jha AK, Lee S, and Lee WJ. Developer mistakes in writing Android manifests: an empirical study of configuration errors. *Proceedings of the 14th International Conference on Mining Software Repositories* 2017; 25-36.

[7] A bug in Yalp Store app - https://github.com/yeriomin/YalpStore/issues/84

[8] A bug in Pinwheel Messenger app - https://github.com/n8fr8/gilgamesh/issues/13

[9] Hu C and Neamtiu I. Automating GUI testing for Android applications. *Proceedings of the 6th International Workshop on Automation of Software Test* 2011; 77-83.

[10] Felt AP, Wang HJ, Moshchuk A, Hanna S, and Chin E. Permission Re-Delegation: Attacks and Defenses. *USENIX Security Symposium* 2011.

[11] Davi L, Dmitrienko A, Sadeghi AR, and Winandy M. Privilege escalation attacks on android. *Information Security* 2011; 346-360.

[12] Wasserman AI. Software engineering issues for mobile application development. *Proceedings of the FSE/SDP workshop on Future of software engineering research* 2010; 397-400.

[13] Zein S, Salleh N, and Grundy J. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software* 2016; 117: 334-356.

[14] Chen PP, Thalheim B, Wong LY. Future directions of conceptual modeling. *Conceptual modeling* 1999; 287-301.

[15] ICCMATT. https://github.com/HiFromAjay/ICCMATT

[16] Jha AK, Lee S, Lee WJ. Modeling and test case generation of inter-component communication in android. *Proceedings of 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)* 2015; 113-116.

[17] Intent and Intent Filters. http://developer.android.com/guide/components/intents-filters.html

[18] Dalvik Virtual Machine. https://developer.android.com/guide/appendix/glossary.html

[19] Enck W, Ongtang M, McDaniel P. Understanding android security. *IEEE security & privacy* 2009; **1**: 50-57.

[20] Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, and Glezer C. Google android: A comprehensive security assessment. *IEEE Security & Privacy* 2010; **2**: 35-44.

[21] Jha AK and Lee WJ. Analysis of Permission-based Security in Android through Policy Expert, Developer, and End User Perspectives. *Journal of Universal Computer Science* 2016; **22**(4): 459-474.

[22] Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM* 2014; **57**(3): 99-106.

[23] Chan PP, Hui LC, Yiu SM. Droidchecker: analyzing android applications for capability leak. *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks* 2012; 125-136.

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

[25] Octeau D, Luchaup D, Dering M, Jha S, McDaniel P. Composite constant propagation: Application to android inter-component communication analysis. *Proceedings of the 37th International Conference on Software Engineering (ICSE)* 2015.

[26] Avancini A, Ceccato M. Security testing of the communication among Android applications. *8th International Workshop on Automation of Software Test (AST)* 2013; 57-63.

[27] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Traon YL, Octeau D, and McDaniel P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* 2014; **49**(6): 259-269.

[28] Kantola D, Chin E, He W, and Wagner D. Reducing attack surfaces for intra-application communication in android. *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* 2012; 69-80.

[29] Ko M, Seo YJ, Min BK, Kuk S, Kim HS. Extending UML Meta-model for Android Application. *Proceedings of the IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS)* 2012; 669-674.

[30] Balagtas-Fernandez FT, Hussmann H. Model-driven development of mobile applications. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (AES)* 2008; 509-512.

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

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

[33] Jing Y, Ahn GJ, Hu H. Model-based conformance testing for android. *Proceedings of the 7th International Workshop on Security (IWSEC)* 2012; 1-18.

[34] Johnson J, Henderson A. Conceptual models: begin by designing what to design. *Interactions* 2002; **9**(1): 25-32.

[35] Takala T, Katara M, Harty J. Experiences of system-level model-based GUI testing of an Android application. *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)* 2011; 377-386.

[36] Amalfitano D, Fasolino AR, Tramontana P. A gui crawling-based technique for android mobile application testing. *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* 2011; 252-261.

[37] Yang W, Prasad MR, Xie T. A grey-box approach for automated GUI-model generation of mobile applications. *Proceedings of International Conference on Fundamental Approaches to Software Engineering* 2013; 250-265.

[38] Jensen CS, Prasad MR, Møller A. Automated testing with targeted event sequence generation. *Proceedings of the 2013 International Symposium on Software Testing and Analysis* 2013; 67-77.

[39] Amalfitano D, Fasolino AR, Tramontana P, Ta BD, and Memon AM. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 2015; **32**(5): 53-59.

[40] Sasnauskas R, Regehr J. Intent fuzzer: crafting intents of death. *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)* 2014; 1-5.

[41] Duran JW, Ntafos SC. An evaluation of random testing. *IEEE Transactions on Software Engineering* 1984; (4): 438-444.

[42] Yang Z, Yang M, Zhang Y, Gu G, Ning P, and Wang XS. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* 2013; 1043-1054.

[43] Instrumentation Test Runner. https://developer.android.com/reference/android/test/InstrumentationTestRunner.html

[44] The GraphML File Format. http://graphml.graphdrawing.org/

[45] Abstract Syntax Tree. http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/

[46] Context. http://developer.android.com/reference/android/content/Context.html

[47] GraphML reader and writer library. https://github.com/tinkerpop/blueprints/wiki/GraphML-Reader-and-Writer-Library

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

[49] Gephi - The Open Graph Viz Platform. http://gephi.github.io/

[50] yEd - Graph Editor. http://www.yworks.com/en/products/yfiles/yed/

[51] Barros P, Just R, Millstein S, Vines P, Dietl W, and Ernst M. D. (2015, November). Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). *30th IEEE/ACM International Conference on Automated Software Engineering* 2015; 669-679.

[52] Li L, Bissyandé TF, Octeau D, and Klein J. Droidra: Taming reflection to support whole-program analysis of android apps. *Proceedings of the 25th International Symposium on Software Testing and Analysis* 2016; 318-329.