Communication-Based Attacks Detection in Android Applications

Chuan Ma, Tao Wang*, Limin Shen*, Dongkui Liang, Shuping Chen, and Dianlong You

Abstract: The Android operating system provides a rich Inter-Component Communication (ICC) method that brings enormous convenience. However, the Android ICC also increases security risks. To address this problem, a formal method is proposed to model and detect inter-component communication behavior in Android applications. Firstly, we generate data flow graphs and data facts for each component through component-level data flow analysis. Secondly, our approach treats ICC just like method calls. After analyzing the fields and data dependencies of the intent, we identify the ICC caller and callee, track the data flow between them, and construct the ICC model. Thirdly, the behavior model of Android applications is constructed by a formal mapping method for component data flow graph based on Pi calculus. The runtime sensitive path trigger detection algorithm is then given. Communication-based attacks are detected by analyzing intent abnormality. Finally, we analyze the modeling and detection efficiency, and compare it with relevant methods. Analysis of 57 real-world applications partly verifies the effectiveness of the proposed method.

Key words: Android; inter-component communication; intents; component hijacking; attack detection

1 Introduction

An enormous number of apps have been developed for Android in recent years, making it one of the most popular mobile operating systems[1]. However, the quality of the booming number of apps can be a concern. Poorly engineered apps may contain security vulnerabilities, which pose a threat to the safety of users.

Of the previously reported app vulnerabilities,

- Chuan Ma, Limin Shen, Dongkui Liang, and Dianlong You are with the School of Information Science and Engineering, Yanshan University, the Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province, Qinhuangdao 066004, China. E-mail: tianyi.mc@126.com; shenllmm@sina.com; cspldk@163.com; youdianlong@sina.com.
- Tao Wang is with the School of Business Administration, Hebei Normal University of Science and Technology, Qinhuangdao 066004, China. E-mail: yy_mma@126.com.
- Shuping Chen is with the Library of Yanshan University, Yanshan University, Qinhuangdao 066004, China. E-mail: Spchen@ysu.edu.cn.

* To whom correspondence should be addressed.

Manuscript received: 2018-10-17; accepted: 2018-11-10 there is a category known as component hijacking vulnerabilities. These component hijacking vulnerabilities are very common and highly threatening, and they can cause security problems such as privilege escalation[2], intent spoofing[3], permission redelegation[4], content leaks and pollution[5], and component hijacking[6].

Prior research has attempted to address some of the above challenges. For example, some analyses of Android applications[4, 7, 8] have been made, but they are largely focused on analyzing application components in isolation. These methods cannot analyze the inter-component communication edges, and therefore are not able to cope with communication-based attacks. Recent works have attempted to expose and analyze the interfaces provided by components to permit interaction[3, 9], but have done so in imprecise ways. For example, Fan et al.[10] proposed a secure mutual authentication protocol to ensure mobile user account security and privacy. A large body of work[6, 11–14] focuses on performing analyses of Android applications for security by transforming the vulnerability detection problem into an equivalent data flow analysis problem.
These prior works have all inspired the present study. In addition, there is a huge number of Android versions in use on a wide variety of hardware, and repairing vulnerabilities in applications involves software developers and users. Even if the vulnerabilities are found in timely manner, there is still great difficulty involved in repairing them in time. Therefore, real-time monitoring of the triggering of sensitive paths and exploitation of vulnerabilities is an effective solution to ensure the safety of Android applications.

In this paper, we consider how applications can prevent these kinds of communication-based attacks. Our contributions are summarized in the following four points.

1) A method of automatically mapping Android Packages (APK) into process algebra is proposed. The complex software systems’ behaviors are abstracted into the relatively simple subsystems’ behaviors and their relationships. Moreover, the algebraic properties of process algebra are used to calculate and reduce the state space of the model to reduce its complexity. The state space explosion problem in behavior modeling is thus effectively solved.

2) We devise an approach that is able to analyze Inter-Component Communication (ICC) edges. That is to say, our approach treats ICC just like method calls, and acknowledges that both control and data can flow on the edges. In this way, important intent data flows can be captured according to inherent Android properties.

3) A runtime sensitive path trigger detection algorithm is proposed that can detect whether the sensitive path is triggered, and detect component hijacking attacks by analyzing intent abnormality. We convert the detection problem to a decision problem of whether a given path is included in a directed graph. The algorithm has a linear time and space complexity, achieves a good level of accuracy and efficiency, and is suitable for practical use. This method does not change the original program in the form of third party applications that detect app behaviors. In the detection process, our method will allow for users to mark flagged paths as non-sensitive, resulting in improved accuracy and efficiency of detection.

4) The proposed method provides an analysis method to track in real-time specific paths of Android systems. It provides a communication-based attack detection method for users to protect the application system, and also a convenient way of testing sensitive paths. It can also provide a comprehensive picture of the possible behaviors that manifest during the running of an app.

The remainder of the paper is structured as follows: Section 2 reviews the previous works that are most closely related to ours; Section 3 introduces the component hijacking problem; Section 4 describes our method; Section 5 gives the process of using our approach for security detection; Section 6 presents experimental results and comparisons with other methods; and the paper concludes in Section 7.

2 Related Work

There have been many works applying static analysis to Android security problems[5–8, 11–14]. Below we describe those that are most relevant to this paper. FlowDroid[11, 12] formally models the event-driven life cycle of an Android app in a dummyMain method. It builds a call graph based on Spark/Soot[13], which conducts a flow-insensitive points-to analysis. FlowDroid then conducts a taint and on-demand alias analysis based on the above call graph, using IFDS[14, 15] which is flow- and context-sensitive. However, it does not address ICC. Epicc[16] statically analyzes ICC and uses an IDE[17] framework to solve for ICC call parameters, but does not link the ICC call sources to targets and does not perform dataflow analysis across component boundaries. CHEX[6] uses a different approach to the modeling of the Android environment, by linking pieces of code reachable from entry points (called splits) as a way to discover data flows between the Android application components. Again, however, it does not address data flow through ICC. Amandroid[18] introduces component-level models (instead of FlowDroid’s whole app-level model) and computes the call graph at the same time as the dataflow analysis by computing the flow- and context-sensitive points-to facts; thus its call graph is more precise, which could lead to fewer false positives in the final analysis results, but it has a higher computing cost than FlowDroid. This paper focuses on critical path detection and real-time detection and therefore adopts FlowDroid, with the added capability of calculating all objects points-to information in a both flow- and context-sensitive way.

Dynamic analysis consists of analyzing applications while they are running. TaintDroid[19] performs dynamic taint tracking on Android. An extension
to TaintDroid handles implicit flows\cite{20}. Dynamic analyses are limited by the way they interact with the User Interface (UI); SmartDroid\cite{21} tackles this issue by combining static and dynamic analyses. We address this problem using static analysis and modeling of the component.

Enck et al.\cite{9} investigated how broadcast intents can leak information and how information can be injected into Receivers. DidFail\cite{22} introduces an approach for tracking data flows between Android components, but the precision of this approach is lower in inter-component path matching. IccTA\cite{23} identifies inter-component privacy leaks by inter-component taint analysis. It is based on a pre-processing step connecting Android components through code instrumentation, which improves the accuracy of the results but may also cause scalability issues. In addition, there has been many works on ICC analysis\cite{5,18,24-27} that are more closely related to ours but still differ in many respects. Our work aims to perform detection at runtime and, moreover, looks to balance accuracy and efficiency.

3 Component Hijacking Problems

The design idea of Android application development is to treat everything as a component. Consistent with this, app developers organize their code into individual application components, each of which completes a logically independent task or provides services for other components. The advantages of this lie in the reduced coupling and increased reusability of modules. The Android operating system also provides a rich inter-component communication method to promote collaboration between applications. The Android ICC has greatly reduced the developer burden and promoted the reuse of modules.

However, the Android ICC also presents security risks; for instance, component hijacking problems have been noted in the literature\cite{2,4}. Messages being passed between components can be sniffed, tampered, or intercepted by malicious applications, leading to negative consequences, such as violations of user privacy and application security policies. For instance, if an app sends data to the wrong recipient, then it might leak sensitive data. Many of the threats present in smartphones are the result of interactions between application components, not just artifacts of single components.

The communication among activity, service, and broadcast receiver relies on the intent, and can be a one-way or a two-way street. The intent is a message that declares a recipient and optionally includes data, and can be thought of as a self-contained object that specifies a remote procedure to invoke and includes the associated arguments. Applications use intents for both inter-application and intra-application communication. If an application provides the name of the receiver component appointed by the intent, the intent is explicit. Otherwise the intent is implicit, and the system will choose the appropriate recipient according to the other parameters of the intent; in other words, the name of the receiver component is anonymous.

An implicit intent presents the risk of a hijacking attack. An application may end up sending intents to the wrong application, which can cause a leak of user information. Data can be stolen by eavesdroppers and permissions can be unintentionally transferred between applications. If an application has vulnerabilities, and its component is unintentionally made public, then external applications can invoke its components in surprising ways or inject malicious data into it.

In general, apps with component hijacking vulnerabilities are not malicious applications, but can be used to perform malicious behaviors by other malicious programs. There are two main defensive approaches: one is to look for the vulnerabilities of applications, another is to detect suspicious behaviors by applications. Our work follows the second approach.

Motivating example. To motivate the research and illustrate our approach, we provide an example of a vulnerability pattern related to ICC among Android apps.

There is a navigation application that obtains the device location (GPS data) in one of its components and then sends it to another component of the app via intra-app intent messaging. The intent involving the location data (Fig. 1, L7–L9), instead of explicitly specifying the receiver component, implicitly specifies it through declaring a certain action to be performed by that component. The second component’s vulnerability occurs on L18 of Fig. 1, where SendSmsActivity uses the system-level API SmsManager, resulting in a message sent to the phone number previously retrieved from the intent. This is a reserved Android API that requires special access permissions to the system’s SMS service. Although SendSmsActivity has that permission, it also needs to ensure that the sender of the original intent message has the required permission.
to use the SMS service.

Due to these vulnerabilities, a malicious app can send the device location data to the desired phone number via text message, without the need for relevant permissions. As shown in Fig. 1, the malicious app first hijacks the intents containing the device location info from the first component. It then sends a fake intent to the second component, containing the GPS data and adversary phone number as the payload. While the example of Fig. 1 shows an exploitation of vulnerabilities in components, it illustrates the general type of attack that may occur by exploiting component vulnerabilities within either a single app or multiple apps.

4 Constructing the Behavior Detection Model

Our analysis is based on FlowDroid, which is a context-, flow-, field-, and object-sensitive and lifecycle-aware static taint analysis tool for Android applications. FlowDroid, based on Soot and Heros, uses a very precise call graph which helps us to ensure flow- and context-sensitivity. Using the tools developed by their own project team, SuSi generates a source and sink list, where source represents a piece of sensitive data (e.g., the device ID) and sink represents a method that may leak data, such as sending a text message. For every sink in the call graph, we can use both forward and backward analysis to propagate access paths to the source, and mark it as (Source, Sink).

In what follows, we provide several definitions for convenience.

- **Definition 1: Critical events.** These are events containing at least one source or sink event.
- **Definition 2: Critical paths.** These are paths starting with the source and ending with the sink (Source, Sink).

We do not need to analyze all the data flows along all possible paths, as only the critical paths can lead to privacy leaks. However, if we only analyze the critical paths, much data dependency information will be lost. Therefore, to balance the accuracy and efficiency of analysis, we must analyze the paths which are related to the critical path. These are made up of the component lifecycle path, the critical event paths, and the communications path between components. Figure 2 illustrates the architecture of our approach.
We obtain the Dalvik Executable (DEX) bytecode file, resources file, and configuration file (Manifest.xml) by reverse-analyzing the APK of Android apps. Using the Soot Dexpler plugin, the DEX bytecode is converted to Jimple, the analysis of which provides Component Event Data Flow Graphs (CEDFG). In this case, the component event refers to the callback function and event associated with the component. Expanding from FlowDroid and using the resources file, Manifest.xml and Google’s official Android documentation, the Component Lifecycle Data Flow Graphs (CLDFG) are also constructed. Merging the CEDFG and CLDFG gives us the Component Data Flow Graphs (CDFG). The ICC model is constructed using the resources file, Manifest.xml, and cross-border data dependency to track the ICC caller and callee. Based on the extended Pi calculus, the behavior detection model is then formally derived. By reverse-analyzing the APK of Android apps, we use an extension of the Pi calculus to express this structure (if \( D \) lastKnownLocation is itself, namely, there is complement action of \( D \) and does not need to be executed synchronously. Such an extended Pi calculus can describe the mobility, external interface communication, and internal state migration, suited to formalizing the critical path of Android applications and ICC.

Trace equivalence and mutual simulation are the two most widely used measures of process equivalence. The choice between them must be based on the functional requirements of the system to achieve a reasonable strategy. Since the purpose of this paper is to detect the runtime behavior to find security issues such as component hijacking, equivalence relations are based on the behavior trace; in other words, if two processes have the same behavior trace, they are equivalent.

**Definition 3: Behavior trace.** If the process has the state migration sequence \( P \xrightarrow{a_1} P_1 \cdots \xrightarrow{a_{n-1}} P_{n-1} \xrightarrow{a_n} P_n \), then the action sequence \( \psi = a_1.a_2.\ldots.a_n \) is a behavior trace of the process \( P \). The set of all possible behavior traces is denoted by \( \text{traces}(P) \).

**Definition 4: Process equivalence.** The different processes \( P \) and \( Q \) are trace equivalent, if and only if \( \text{traces}(P) = \text{traces}(Q) \).

**Definition 5: Alpha conversion.** When the name of a process is converted to a new name; this is called alpha conversion.

In general, a replacement new name should not exist in the process. If the name exists, the name must be changed to ensure no naming conflict. We use the symbol \( \{b/a\}P \) to express the name \( a \) in \( P \) being replaced by \( b \). For example, if \( P = vba.b \), then \( \{c/a\}P = vbc.b.\{c/b\}P = vca.c.\{b/a\}P = vb\{b/b\} \).

\[
\begin{align*}
(1) \text{INP:} & \quad \frac{x(z).P \xrightarrow{x(y)} \{y/z\}P}{x(z).P \xrightarrow{x(y)} \{y/z\}P} \\
(2) \text{TAU:} & \quad \frac{\tau.\tau \xrightarrow{\tau} \{y/z\}P}{\tau.\tau \xrightarrow{\tau} \{y/z\}P}
\end{align*}
\]

(3) \( P_1 || P_2 \) means that if action \( (x) \) in \( P_1 \) and co-action \( (\overline{x}) \) in \( P_2 \) are subordinated to synchronous set \( A \), then \( x \) and \( \overline{x} \) execute synchronously, and the action is expressed with \( r(x) \) after the synchronous execution, while the general actions \( \lambda \) are executed asynchronously.

(4) The restriction operator \( \nu a P \) means that the domain of the name \( a \) is limited to the process \( P \).

(5) \( !P \) stands for the replication process \( P \). The replication operators can define the recursive processes, and they also can describe the process of creating a runtime instance of an application or component.

The syntax and semantic specification of the extended Pi calculus are as follows.

\[
P :: := \sum_{i \in I} a_i.P_i | P_1 || P_2 | \nu a P || !P.
\]

4.1 Extended Pi calculus Since the process algebra\(^{(25,26)}\) can effectively describe the distributed and message communication features of the Android architecture, we use an extension of the Pi calculus\(^{(27)}\) as our modeling language. The syntax and semantic specification of the extended Pi calculus are defined as

\[
P :: := \sum_{i \in I} a_i.P_i | P_1 || P_2 | \nu a P || !P.
\]
(3) SUM-L: 
\[ \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}; \]

(4) COMM-L: 
\[ \frac{P \xrightarrow{\pi(y)} P', Q \xrightarrow{x(z)} Q'}{P || Q \xrightarrow{\tau(x)} (y/z)(P' || Q')}; \]

(5) CLOSE-L: 
\[ \frac{P \xrightarrow{\pi(y)} P', Q \xrightarrow{x(z)} Q'}{P || Q \xrightarrow{\tau(x)} \nu z \{y/z\}(P' || Q')}; \]

(6) REP-ACT: 
\[ \frac{\nu P \xrightarrow{a} P'||P}{!P \xrightarrow{a} P'||P}; \]

(7) REP-COMM: 
\[ \frac{\nu P \xrightarrow{\pi(y)} P', P \xrightarrow{x(z)} P''}{!P \xrightarrow{\tau(x)} \{y/z\}(P' || P'')}!P. \]

4.2 Generating CDFG

The following four steps are used to generate component data flow graphs.

4.2.1 Generating CLDFG

Unlike standard Java programs, Android applications do not have a main method. In Android, numerous types of events (e.g., system events and UI events) can trigger callback methods defined in the app, meaning that an app can have many entry points. This makes it difficult to generate a control flow graph for Android apps.

FlowDroid generates a new dummyMain method for each app analyzed. Each main method will only involve the parts of the lifecycle that, according to the app’s XML configuration files, can actually occur at runtime. Disabled activities are automatically filtered by the runtime. Disabled activities are automatically filtered among components, such as component hijacking.

\$i0 in Fig. 3 is a local variable in the stack that saves state information of the component lifecycle. For instance, $i0 == 3 means an activity is running, and the system is in the onResume method; $i0 == 4 means an activity is no longer visible, and the system will invoke the onPause method. The state information is difficult to evaluate statically, so we treat it as an opaque predicate and represent it by $p$.

Unfortunately, FlowDroid does not handle ICC and cannot address security issues involving intent passing among components. Since the goal of this paper is to detect security problems involve with communication among components, such as component hijacking,
If it jumps up, then we remove the edge (as shown the edge marked by $\times$ and ①③ on Fig. 4), and replace it with a new edge pointing to the nearest onCreate that is above goto, as shown in the edges marked ① and ③ on Fig. 4.

Using the above algorithm to split the dummyMain method, we add the component lifecycle methods and UI callback methods to the figure; we then get each CLDFG. On this basis, we can work out the component lifecycle data flow graphs using the data flow analysis methods presented in next section, as shown on the left side of the part of each component in Fig. 5.

In order to precisely know the implementation method of a virtual method invocation, we need to know the receiver object’s dynamic type, while flow-sensitive data flow analysis requires us to know how the program control flows. There is a mutual dependency between these two analyses, thus we need to create a data flow graph for each component.

4.2.2 Generating CEDFG

We have got the component lifecycle model. Once the critical events in lifecycle are triggered, it can lead to the spread of sensitive information. Therefore, we need to get the critical event data flow graphs.

Drawing on the motivating example, based on the method of FlowDroid, we can find a sink (startActivity (Android.content.intent)) in the onCreate method, and a corresponding source (getLastKnownLocation in onCreate), so onCreate is a critical event. Whereupon we use Soot to analyze the onCreate method, and convert an app’s Dalvik bytecode to Jimple. This is shown in Fig. 6.

---

**Fig. 4** Components lifecycle by splitting dummyMain method.

**Fig. 5** Component data flow graphs of motivating example and ICC model.
Jimple is a compact, no stack, typed address code Intermediate Representation (IR) provided by Soot. Each instruction in Jimple has an explicit operand, making it suitable for process analysis and code optimization.

The temp variables in Jimple start with temp$, which can only be assigned once, and are unique, so they can be a reference with constancy to express data dependencies. We record variable declaration statements into the data-type table and record the creation site of temp variables into the temp-creation table.

Jimple has fifteen kinds of expression statements. Some are used in the process control flow, such as IfStmt, GotoStmt, TableSwitchStmt, and LookupSwitchStmt, and some are used for inter-procedural control flow, such as InvokeStmt, ReturnStmt, and ReturnVoidStmt. In the CEDFG, we only retain these two kinds of statements as the basic block. Therefore, compared with the traditional method, the data flow graph constructed by our method has a smaller basic block. The CEDFG has edges which link data dependencies. This is shown in Fig. 7.

To analyze data dependencies in the Jimple file, we keep track of two kinds of information, defined as follows.

**Definition 6: Object characteristic.** If there exists a statement such as $x = new Object(\ldots)$ in the Jimple, then $x$ is object. The object characteristic is composed of fields, methods, and connectors. It is shown by a four-
tuple, OC={object, fields, methods, connector}, where connector={.., ::} is same as the connector definition of C++, by which “.” denotes a member-selection operator and “::” denotes a domain identifier. In order to facilitate identification of the object type, we use the connector link object, fields, and methods, such as flonCreate() and i1.mAction in Fig. 5.

- Definition 7: Data fact. Each data fact is denoted as (v, t), where v is the variable (whose type is an object reference type) and t is a constant, a temp variable, or a mixture of both.

For example, the data fact (i1, temp$0) means that variable i1 points to temp$0, which is an intent object as seen from the data-type table.

In traditional dataflow analysis methods, the sites of use or creation of objects and variables are labeled by code line numbers. In contrast, our method uses temp variables to label this information. Compared with the traditional methods, temp variables permit more information because they are a declared data type in Jimple.

Data facts describe the data flow information of components. Through analyzing the relationship among data facts, data dependencies can be obtained, as shown in Fig. 7.

In Fig. 7, the data-type table and the temp-creation table hold all the information necessary to analyze the data dependencies. Following the classical static analysis approach[28], the general procedure of data dependencies analysis in our method is as follows.

Firstly, we get the data dependencies. For convenience, the entry and exit sets of the basic block b are respectively denoted by entry(b) and exit(b). If there is a data fact (v, t) ∈ entry(b) and there is another data fact (v, t) ∈ exit(b) with the same variable v, then we replace the variable v with the temp variable t, thus replacing (v, t) with (t, t'). The data facts to be replaced are called data dependencies. For example, we replace i1 with temp$0 in data fact (i1.mAction, "showlocation") (as shown in Fig. 7) and arrive at the data dependency (temp$0.mAction, "showlocation"). For the intent object i1, we make two other replacements and arrive at the data dependencies (temp$0.mExtras, "network", temp$4) and (temp$0.mExtras, "PHONE_NUM", temp$7). In this way, we get dependencies of data in the flow which are related to the temp$0.

Secondly, we get additional information about the data with the help of the data-type and temp-creation tables. By retrieving the information associated with i1 in these tables, we know that i1 specifies "showlocation" as its start action and fills its mExtras field with the “network” and “PHONE_NUM” (the values of which are temp$4 and temp$7, respectively).

Finally, we track the sensitive data. By analyzing the temp-creation table, we find the sensitive API “getLastKnownLocation” and its return value assigns to temp$3, as shown in Fig. 7. Using this API you can obtain location information, so we know that temp$3 is sensitive data. By analyzing the data dependencies related to temp$3 ((temp$5, temp$3), (temp$4, temp$5), and (temp$0, mExtras, (“network”, temp$4))), we know that temp$3 is filled to the mExtras field of i1 via temp$5 and temp$4. Therefore, intent i1 is an object with sensitive data. Through data dependencies analysis, we get the transmission path of sensitive data in the component; the analysis of sensitive data across component boundaries will be introduced in the next section.

After data flow analysis, we can get the data flow graph in Fig. 6, which highlights the relevant part of each component. The nodes in the data flow graph are Android APIs or custom methods, and the annotations at the edge are data dependencies. The ‘::’ behind each method is the return value of the method.

4.2.3 Combining the data flow graphs

Critical paths consist of component lifecycle paths that involve critical events and inter-component communication. We use the data flow analysis methods in the previous section and analyze the component life cycle control flow of the motivating example, arriving at the component lifecycle flow graph. Next we merge the two parts (CLDFG and CEDFG) to get the CDFG.

Assuming vi and vj is the vertexes of lifecycle data flow graph (CLDFG_{com1} = {LV_{com1}, LE_{com1}}), namely, vi, vj ∈ LV_{com1}, eij ∈ LE_{com1} is the direction edge connecting vi and vj. If the events in vi is critical events, vm and vn are the CEDFG entrance and return points, respectively. The combining methods are as follows:

1. Add directed edge eim to connect points vi and vm. Add directed edge enj to connect points vn and vj, marked by red edges ① and ②③④ in Fig. 5.

2. Delete the directed edge eij, marked by × in Fig. 5.

Figure 5 is a data flow graph of the components named FindLocationActivity and SendSmsActivity in
motivating example. The left part of each component is its lifecycle data flow graph, and the right part is a data flow graph for the critical events. The two components of the motivating example both have only one critical event, namely onCreate, and their corresponding (Source, Sink) is (getLastKnownLocation ("network"), startActivity(i1)) and (getIntent(), sendTextMessage(number, message)), marked by a bold red frame in Fig. 5. The path of (Source, Sink) is the critical path. After using the above method to combine the two graphs, we arrive at the component data flow graphs.

4.2.4 Modeling of ICC

We can know whether the intent communications between components is likely to have been intercepted or tampered with. To identify such security problems, an analyzer needs to be aware of control and data flows across component boundaries; this requires a number of steps as follows.

(1) Extract the fields of an intent object. The destination of an ICC can be either explicitly or implicitly specified in the outgoing intent. Table 1 lists some frequently-used methods of explicit intent and implicit intent, and fields that can be manipulated by invoking these methods. The common way of creating an explicit intent is adding the destination component’s name (by mComponent) using Android APIs such as setClass or a special constructor for the intent. An implicit intent requests a general action (by mAction) to perform, and the system finds a capable component which can fulfill the request.

We can extract the fields of an intent object in accordance with Table 1. For instance, we can get the data dependency \( temp$0,mAction, "showlocation" \) from the statement \( i1.SetAction ("showlocation") \) in Fig. 5.

(2) Find the ICC caller and callee. We find out all possible ICC callers and callees, as shown in Table 2, but whether the ICC caller and callee communicate depends on further analysis of the fields of the intent.

For an explicit intent, we can find the ICC caller and the corresponding callee according to mComponent. But for an implicit intent, the Android system finds the destination depending on the intent fields as well as the manifests of all the apps which specify intent filters for a component. An intent filter is an XML expression made up of an action tag, category tag, and data tag (which includes both uri and type). The Android system determines the destination of an implicit intent by

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Part of analysis data in the intent communication.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent type</td>
<td>Method</td>
</tr>
<tr>
<td>Explicit intent</td>
<td>intent(Context, Class(??))</td>
</tr>
<tr>
<td></td>
<td>putExtras(Bundle/intent)</td>
</tr>
<tr>
<td></td>
<td>getIntent()</td>
</tr>
<tr>
<td>Implicit intent</td>
<td>putExtra(String, ...)</td>
</tr>
<tr>
<td></td>
<td>setClassName(Context, String)</td>
</tr>
<tr>
<td></td>
<td>startActivity(Intent)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Part of ICC caller and callee.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC caller</td>
<td>ICC callee</td>
</tr>
<tr>
<td>sendBroadcast(intent i)</td>
<td>onReceive(Context context, intent intent)</td>
</tr>
<tr>
<td>sendBroadcast(intent i, String rcvrPermission)</td>
<td></td>
</tr>
<tr>
<td>sendOrderedBroadcast(intent i, String recvrPermission, BroadcastReceiver receiver, ...)</td>
<td></td>
</tr>
<tr>
<td>sendOrderedBroadcast(intent i, String recvrPermission)</td>
<td></td>
</tr>
<tr>
<td>sendStickyBroadcast(intent i)</td>
<td>getPendingIntent()</td>
</tr>
<tr>
<td>sendStickyOrderedBroadcast(intent i, BroadcastReceiver receiver, ...)</td>
<td></td>
</tr>
<tr>
<td>startActivity(intent i)</td>
<td>setResult(int resultCode, Intent intent)</td>
</tr>
<tr>
<td>startActivityForResult(intent i, int requestCode)</td>
<td></td>
</tr>
<tr>
<td>onActivityResult(int requestCode, int resultCode, Intent intent)</td>
<td></td>
</tr>
<tr>
<td>startService(intent i)</td>
<td>onStartCommand(Intent it, int flags, int startId)</td>
</tr>
<tr>
<td>bindService(intent i, ServiceConnection conn, int flags)</td>
<td>onBind(Intent intent)</td>
</tr>
<tr>
<td></td>
<td>onUnbind(Intent intent)</td>
</tr>
</tbody>
</table>
applying a set of rules matching the relevant intent fields and the intent filter specification for every component on the system. The PackageManager has a set of query...() methods that return all components that can accept a particular intent, and a similar series of resolve...() methods that determine the best component to respond to an intent.

We run a precise action test, category test, and data test (having both uri and type) to find the destination component(s).

Taking the example in Fig. 5, we can discover an ICC caller (startActivity(i1) of FindLocationActivity) and callee (getIntent() of SendSmsActivity) by the data dependency \(\{\text{temp}\!0.\text{mAction}, \text{"showlocation"}\}\) and the intent filter information of SendSmsActivity (\textit{action Android : name = "showlocation"}).

(3) Track data flow from the ICC caller to callee. In Fig. 5, for the data flow \(f1 :: \text{startActivity}(i1) \rightarrow ss :: \text{getIntent}() : \text{temp}\!0 \rightarrow \{i2, \text{temp}\!0\}\), we can get the data dependencies \(\{ss :: i2, f1 :: i1\}\) between components marked by the red edge \(5\). It is easy to track the data flow from the ICC caller to callee by the data dependencies.

4.3 Formalized behavior detection model

In order to obtain the formalized behavior detection model, the various elements in the CDFG are mapped to the process expression of the extended Pi calculus. The mapping rules are given below.

4.3.1 Action mappings

According to the syntax of the extended Pi calculus, the prefix action is defined as \(a ::= \lambda [\pi ([p])]\). The different mappings are made according to the different action properties, as shown in Fig. 8.

The process identification is represented by the base block number. In general, there is a guard relationship between the action in the current base block and the next basic block number; that is to say, the next base block is executed only after the action in the current base block has been executed. Therefore, it is mapped to a process guard like \(P = a.Q\), as shown in Fig. 8a. In Fig. 8b, the concurrent action record mark \(\tau(x)\) represents the actions completed concurrently. The synchronization occurs only when process \(P\) performs a synchronous action \(x\) and process \(Q\) performs the corresponding complement action \(\overline{x}\). In accordance with Fig. 8a, the synchronous actions are mapped to the process guard \((P = x.P'\text{ and } Q = \overline{x}.Q')\), then the complementary actions \((x, \overline{x})\) are added to the

synchronous set \(A\). Moreover the synchronous actions are replaced with \(\tau(x)\) in the algebraic calculus. The fuzzy conditional decision action occurs in the component lifecycle model, as shown in Fig. 8c. Since it is impossible to determine the trigger sequence of each callback function, we introduce nondeterminacy, which maps the trigger action to the unobservable action \(\tau\) and connects each branch to a sum process with an alternative operator +. However, the determination condition decision action occurs in other models, such as callback functions, as shown in Fig. 8d. Because our model distinguishes true assertions ([\(P = true\)]) and false assertions ([\(P = false\)]) by analyzing Jimple sentences to determine the branch conditions, it joins them into a sum process with an alternative operator +.

4.3.2 Inter-procedure call mapping

An inter-procedure call will cause the main procedure to interrupt execution. After the interrupt address is recorded and the field information is saved, the sub-procedure of the call is executed instead. When the sub-procedure is finished, it will jump to the interrupt address of the main procedure and continue executing. There are two key factors to consider for mapping. (1) When the main procedure calls the sub-procedure, it needs to know the sub-procedure’s location explicitly. Otherwise, the data flow graph will have unreachable vertices. (2) The number of the first basic block in the sub-procedure is abstracted as the calling channel.
The process \( P \) of the main procedure calls the sub-procedure. The interrupt address at which the main procedure calls the sub-procedure, corresponding to the number of the basic block, is sent as a parameter to the sub-procedure, and returned as a return channel (corresponding to the return edge) after sub-procedure execution has completed. In all return base blocks, the name can be received only if the interrupt address corresponding to the number is the same as the name of the return channel. The mapping rules for inter-procedure calls are shown in Fig. 9.

In Fig. 9, the addresses of the base block \( P \) and \( Q \) are represented by the new names \( p \) and \( q \), respectively, that is, the procedure call address and return address. Therefore, \( p \) and \( q \) can be used by caller and callee as channels for calling and returning, respectively. In order to return correctly, the parameter \( p \) is sent to the sub-procedure \( \text{Fun()} \) when the main procedure calls it, namely \( \overline{q}(p) \). The sub-procedure receives \( p \) by \( q \) and stores it in \( l \), which is a return channel (the value is \( p \)) after the sub-procedure execution has completed. When the main procedure is received by \( p \), the procedure call is completed, ensuring that the sub-procedure returns to the normal calling position. This process can be formally described as follows:

\[
P \parallel [Q = v \, q \, (q(l). \cdots \overline{l}.0)] \parallel \nonumber \\
\vdash q(l). \cdots \overline{l}.0) || 
\]

\[
v \parallel q(p/l).p.R || 
\]

\[
\vdash q(l). \cdots \overline{l}.0) || 
\]

The process \( Q \) of sub-procedure \( \text{Fun()} \) may be called by other procedures, so the replication operator \( ! \) is used. In accordance with the operation semantics of the extended Pi calculus, the main procedure and the sub-procedure perform concurrent calculus. After using the migration rules, such as COMM-L and CLOSE-L, concurrent calculus migrates \( P \parallel Q \) to \( R \parallel Q \). That is, after a series of operations \( \tau(q) . . . \tau(p) \), the main procedure of the application calls the sub-procedure completely, and the sub-procedure returns to the initial state and waits for the other procedure call. \( \tau(p) \) and \( \tau(q) \) represent the unobservable actions occurring on the channels \( p \) and \( q \), respectively, and the ellipsis represents a series of operations in the sub-procedure. By this mapping, we use the extended Pi calculus to accurately represent the entire procedure of inter-procedure calls.

### 4.3.3 Behavior detection model

In the Pi calculus, the behavior pattern of the system is described by processes. The behavior detection model of an Android application is represented by a process expression. The process \( P_{\text{App}} \) is composed of each component process \( P_{\text{component}} \), which is composed of the component life cycle process \( P_{\text{comlife cycle}} \) and event process \( P_{\text{event}} \). That is

\[
P_{\text{App}} ::= P_{\text{comlife cycle}} \parallel P_{\text{event}} . 
\]

Therefore, according to the mapping rule, we get the process expression of the FindLocationActivity component and its onCreate event as follows:

\[
P_{\text{App}} = P_{\text{flonCreate}} \parallel P_{\text{flonCreate}} . 
\]

Similarly, the process expression of the SendSmsActivity component and its onCreate event are as follows:

\[
P_{\text{App}} = P_{\text{sslifecycle}} \parallel P_{\text{sslifecycle}} . 
\]

As is apparent, the ICC relationship has not yet been expressed. So we rewrite the process expressions \( P_{\text{flonCreate}} \) and \( P_{\text{sslifecycle}} \) based on the inter-component data dependency \( \langle s s : f l \rightarrow f l \rangle \) as shown below:

\[
P_{\text{flonCreate}} = v \, fr \, (\text{onCreate}(fr)r \, \cdots \, \text{putExtra}. \, \text{putExtra.startActivity}(fr)) \nonumber 
\]

\[
P_{\text{sslifecycle}} = v \, sr \, (\text{onCreate}(sr)r \, \cdots \, \text{sendTextMessage}(sr)) \nonumber 
\]

In the process expression, we add an interactive channel \( fs \) for \( P_{\text{flonCreate}} \) and \( P_{\text{sslifecycle}} \). \( P_{\text{flonCreate}} \) sends \( i1 \) to \( P_{\text{sslifecycle}} \) and replaces \( i2 \) with \( i1 \) by \( fs \). Because
$P_{\text{flonCreate}}$ does not need to return when it calls $P_{\text{ssonCreate}}$. It does not construct a return channel for them as for normal inter-procedure calls.

The process expressions of each component are combined concurrently. The interaction between $P_{\text{flonCreate}}$ and $P_{\text{ssonCreate}}$ is as follows:

$$
\begin{align*}
\rho_{P_{\text{flonCreate}}} = \rho_{P_{\text{ssonCreate}}} = & v \ s (\rho_{P_{\text{flonCreate}}}) \setminus \rho_{P_{\text{ssonCreate}}} \\
\rightarrow & v \ fr \ sr \ (fr(i_1), fr(i_2).setContentView. \cdots \fr) \\
\end{align*}
$$

(1) Extract the target API, and scan the target program’s component data flow graphs to obtain the package name, method name, the value and type of arguments, and return values.

(2) Regard the API extracted in Step (1) as a message to input, with the beforeHookedMethod and afterHookedMethod in Xposed used to get the trigger time of the relevant API, input parameter values, and the return value. Then proceed to analyze data dependencies.

5.2 Get the component lifecycle

(1) Get lifecycle state, by detecting the currently running task, and determining the name of running components and their lifecycle states.

(2) Run Xposed and monitor callback methods of the component’s lifecycle and UI to get its run-time information, such as triggering time, parameters, and return values.

5.3 Detecting the triggered critical paths

If the critical paths are triggered at runtime they are likely to cause privacy leaks, so we detect this and send a notice to users.

By using the method provided in the previous section, we can capture an API sequence for each component at runtime. For each API call we capture the triggering time, parameter values, return value, data dependence, and so on, thus it has uniqueness and can have a one-to-one correspondence with the vertex in the CDFG. This is shown in Fig. 10, in which the paths of the red edge are the critical path.

If the path formed by these API sequences is a part of the whole CDFG and the critical path is included in the path, this shows that the critical path is triggered and sensitive data is exposed. An example of this is shown in Fig. 10, in which the captured API sequence $A \rightarrow B \rightarrow C \rightarrow D$ belongs to CDFG. Since $B \rightarrow C \rightarrow D$ is the critical path, sensitive data may be exposed.

We will change the detection problem to a decision problem of whether a given path is included in a directed graph. We through the following steps to determine this, assuming that a given path is $V_0 \rightarrow V_1 \rightarrow \cdots \rightarrow V_n$, and $G_d$ is the directed graph.

(1) Search the initial vertex. Search $V_0$ in $G_d$, and if it exists, move to Step (2);

(2) Determine if the subsequent vertex is in the directed graph. Search all of the subsequent vertices of $V_0$ in $G_d$. If $V_1$ is found, then move to Step (3), otherwise we can say this given path is not in the directed graph.

Fig. 10 API sequence and the corresponding CDFG.
(3) Iterate the decision process. Replace $V_0$ with $V_1$, $V_2$ with $V_1$, and repeat Step (2) until all the vertices of a given path are analyzed.

We use the adjacency table to store CDFG; the time complexity of the depth-first walk and breadth-first calendar calculation method are both $O(n + e)$, where $n$ represents the number of vertices, and $e$ represents the number of edges. For searching the given vertices in a directed graph, in general the breadth-first time complexity compared with the depth-first walk, but space complexity is high. Detection of attacks pays more attention to time efficiency, so we use a breadth-first calendar calculation method to search the initial vertex. We give the critical path trigger detection algorithm as Algorithm 1.

When using this algorithm for testing, when a critical path is triggered the user will be notified, and at the same time the model will record the path information and data dependency information. For example, when we detect FindLocationActivity in the motivating example, we catch the API sequence: $\text{!getLastKnownLocation( )} \rightarrow \text{toString( )} \rightarrow \text{putExtra( )} \rightarrow \text{putExtra( )} \rightarrow \text{startActivity( )}$, because it contains the critical path, and bring up a prompt as shown in Fig. 11.

5.4 Attack detection

If components $i$ and $j$ in the process of communication suffer from an intent-based attack, then intent may exhibit one of three exceptions:

(1) Intent is intercepted: component $i$ has sent an intent to $j$, but component $j$ has not received it, such as in the case of an intent hijacking attack.

(2) Intent is faked: component $j$ received an intent from $i$, but $i$ did not in fact send any intent to component $j$, such as the case of an activity launch attack.

(3) Intent is tampered with: component $i$ sent an intent to component $j$, but the received intent is different from the one that was sent.

The API sequence of each component is captured at runtime, and the process expressions are migrated according to the API sequence. For example, if an intent hijacking attack occurs, when FindLocationActivity performs the operation startActivity to send the intent, the process expression migration is as follows:

$$\text{!fr sr fr sr fs sr fs sr fs}$$

$$\text{fr sr fr sr fs sr fs sr fs}$$

From this we can see that FindLocationActivity will be returned after the channel fs after sending i1, and the next step of component SendSmsActivity should execute setContentView. Namely,

$$\text{r(i1 i2) fr sr setContentView.$$}

However, the i1 has been hijacked, and the setContentView action is not executed, which is not consistent with the migration of the expression. So the attack is detected.

Finally, we match the intent information of the two components, as shown in Fig. 12. If the intent information of the ICC caller and callee is consistent, the communication is normal. If any of the above
mentioned three kinds of abnormal situations appear, we raise a warning, intercept, and discard the intent. Figure 13 shows the detection results after the HijackActivity runs in the motivating example.

Our method offers real-time monitoring and identification of the critical paths being triggered. It will give a notification if the critical path is executed, and if the path is not threatening to privacy, the user can tag it as such and the method will remove the path from the sensitive path automatically. As the number of the paths marked by users increases, testing efficiency and accuracy improve.

6 Experimentation and Evaluation

In this section, we analyze the modeling and detection efficiency of our method, and compare it with relevant methods. Through analyzing 57 real-world applications, we partly verify the effectiveness of the proposed method.

6.1 Performance evaluation

The most computationally intensive step in our approach is building the CDFG. Once the CDFG is built, the running times of the detection of the critical path trigger and component hijacking attacks are negligible in comparison, with time-space complexity of $O(n^2)$.

For the 57 apps used in the experiment, our approach incurs only a 7% performance overhead on a CPU-bound micro-benchmark and imposes a negligible overhead on interactive third-party applications.

Figures 14a and 14b separately present the time and space taken by our approach to construct the CDFG for 57 real-world apps. For each app, which consists of multiple components, the median time is 128 seconds, with a minimum of 2 seconds and a maximum of 23 minutes and 3 seconds. The median space is 649 KB, with a minimum of 12 KB and a maximum of 6.771 MB. The scatter plots in Fig. 14 show both the running time/space and the size of the app (measured by the number of Jimple instructions).

Figures 15a and 15b separately present the time and space overhead of the detection phase of our approach for the 57 apps. The columns show both the running time/space and the number of apps which are detected. Our approach incurs only a 14.9% performance overhead on a CPU-bound micro-benchmark when the number of the apps detected is 57, with a space overhead of 20.8%.
6.2 Comparison of benchmarks test

Using real-world apps for comparison is difficult since there is no easy way to determine the ground truth. We compare the effectiveness of our approaches data leak detection with the other tools on two benchmarks: DroidBench2 and ICC-Bench. Using our approach, we build the behavior model for each app and conduct the behavior detection in real time. The results are shown in terms of numbers of True Positives (O), False Positives (*), and False Negatives (X). If a test app contains multiple data leak paths, the result is shown for each of them. The DroidBench2 and ICC-Bench test results are shown in Table 3.

6.3 Effectiveness evaluation of detecting attacks

We use the 57 Android apps to evaluate the effectiveness of detecting attacks. The results are listed in Table 4. The values separated by “/” in the cell are numbers of attacks, where the first value is the actual attack count at runtime, the second value is the attack count detected by FlowDroid, the third value is the attack count detected by DidFail, the fourth value is the attack count detected by Amandroid, and the final value is the attack count detected by our method. From the results, we can see one intent tampering, two intent intercept attacks, and one intent counterfeit attack were not detected, giving a success detection rate of 97%. Although only based on 57 test apps, this partly proves the effectiveness of our method.

7 Conclusion

In this paper, we argue that detecting communication-based attacks at runtime is very challenging. We have...
presented an efficient and accurate method to analyze an app’s sensitive path at component level. In this method, we extend the dummyMain method of FlowDroid to the component level, generate data flow graphs for each component, and analyze the data flow across component boundaries. The CDFG is then formally mapped based on the extended Pi calculus, and a formalized Android application behavior detection model is constructed. In accordance with this model, we presented an attack detection method based on analysis of intents. Experimental verification and comparative analysis verify the detection performance and effectiveness of the proposed method. We found the hierarchical characteristics of Pi calculus particularly suitable for the componentization of Android applications, and also suitable for behavior modeling and detection based on the communication patterns. It can effectively deal with communication attacks on the mobile platform. The abstract characteristics of process algebra can help to hide the implementation details in the system, extract the semantics, and obtain the essential abstraction of behavior. Formal features such as reasoning and calculus help to automate behavior modeling and detection. Further research is needed for modeling and detection in the context of concurrent behaviors introduced by technologies such as multithreading and asynchronous calls in Android applications.

Acknowledgment

This research was supported by the Hebei Provincial Natural Science Foundation (Nos. F2016203290 and F2017203307), the National Natural Science Foundation of China (No. 61772450), the Doctoral Foundation of Yanshan University (Nos. BL18011 and B906), the Hebei Normal University of Science and Technology Scientific Research Foundation (No. 2018YB019), the China Postdoctoral Science Foundation (No. 2018M631764), and the Hebei Province Science and Technology Planning Project (No. 17210701D).

References


Chuan Ma is currently an associate professor in the School of Information Science and Engineering, Yanshan University. He received the PhD degree from Yanshan University in 2017. His main research interests are focusing on information security and software formal methods.

Tao Wang is currently an associate professor in Hebei Normal University of Science and Technology. She received the PhD degree from Yanshan University in 2014. Her current research interests are intrusion detection and collaboration computing.

Limin Shen is currently a professor in the School of Information Science and Engineering, Yanshan University. He received the PhD degree from Yanshan University in 2005. He worked in the Department of Computer Science, Illinois Institute of Technology, from 2005 to 2007 as a visiting scholar. His main research interests are focusing on flexible software technology and information security.

Shuping Chen is currently a librarian in the library of Yanshan University. She received the MS degree from Yanshan University, China, in 2008. Her current research interest is intelligence information management.
Dongkui Liang is currently a PhD student at Yanshan University. He received the BS and MS degrees from Yanshan University, in 2003 and 2010, respectively. He is now working as a lecturer in the School of Information Science and Engineering, Yanshan University. His current research interests include flexible software technology and information security.

Dianlong You is currently an associate professor at Yanshan University. He received the PhD degree from Yanshan University in 2014. His current research interests include software intrusion detection and software formal methods.