Check Points against Privacy Breaches in Android Applications

Kazuhide Fukushima[†], Lujo Bauer[‡], Limin Jia[‡], Shinsaku Kiyomoto[†], and Yutaka Miyake[†]

KDDI R&D Laboratories, Inc, JAPAN

Carnegie Mellon University, USA

Summary
The risk of privacy breaches by malicious programs has been increasing, and these programs have used more elaborate techniques to circumvent detection. Attacks using a collaboration of applications are especially difficult to find since distinct applications obtain privacy-sensitive data and send the data to the outside. Current mobile platforms have a security enforcement mechanism based on a sandbox to prevent direct data sharing between applications. Furthermore, several schemes have been proposed to improve the security against the attacks caused by two or more applications that communicate with each other. However, these schemes cannot monitor all the possible data-sharing methods. A security analysis that covers a wider range of possible data-sharing methods between applications is required for protecting leakage of privacy-sensitive information. In this paper, we present a detailed manual analysis regarding a wider range of possible methods for sharing data in the Android OS, and show how to detect actual privacy breaches using existing frameworks. Our analysis contributes to the enhancement for the security of the Android OS.

Key words:
Information-flow analysis, Software Verification, System Security, Software Security, Android Security

1. Introduction

The risk of privacy breaches by malicious programs has been increasing, and these programs have used increasingly elaborate techniques to circumvent detection. Symantec [31] has reported a security issue caused by collaboration of two or more malwares. For example, a malicious program that can obtain privacy-sensitive data interacts with other programs that play the role of sending data outside. We have to track the information-flow of these programs to prevent these privacy breaches.

Existing mobile platforms, such as Android OS, iOS, and Windows 8 have a security enforcement mechanism based on a sandbox to prevent direct data sharing between applications.

Each Android application runs on an instance of Android Dalvik virtual machine that is a distinct sandbox executed as an independent Linux process. If an application needs to access the resources beyond the boundary of the sandbox, the application has to request permissions from the Android OS. An application declares all permissions in the Manifest file, and these permissions are authorized by users at the time of its installation. After the user authorization, the Android OS grants the permissions to the application. Several existing systems that enhance the original security enforcement mechanism of the Android OS can track information-flows between applications; however, the system cannot monitor all the possible data-sharing methods. Accurate detection of the information leakages to establish a more accurate detection mechanism of privacy breaches. Accurate detection of privacy breaches requires fine granularity information-flow analysis.

In this paper, we study a wide range of possible cases of data-sharing between components of Android applications and extract the check points for them. Then, we consider concrete detection mechanisms of privacy breaches based on these check points. Our results contribute to enhance the current security enforcement mechanism in Android OS.

2. Related Work

Information-flow tracking systems have been proposed for source code in C and Java, or binary programs [6, 7, 24, 17, 21, 10, 18, 8, 30, 29, 32, 26]. These systems can be used to detect malware that leaks privacy-sensitive data to third parties, as well as memory corruption attacks, e.g., buffer overflow attacks against programs implemented in C.

```
public class Main extends Activity{
  @Override
  public void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    final SharedPreferences pre = getSharedPreferences
       ("Main", MODE_WORLD_READABLE);
    final Button button = (Button)findViewById
       (R.id.button);
    button.setOnClickListener(new OnClickListener(){
      @Override
      public void onClick(View view){
        Editor editor = pre.edit();
        EditText editText = (EditText)findViewById
```

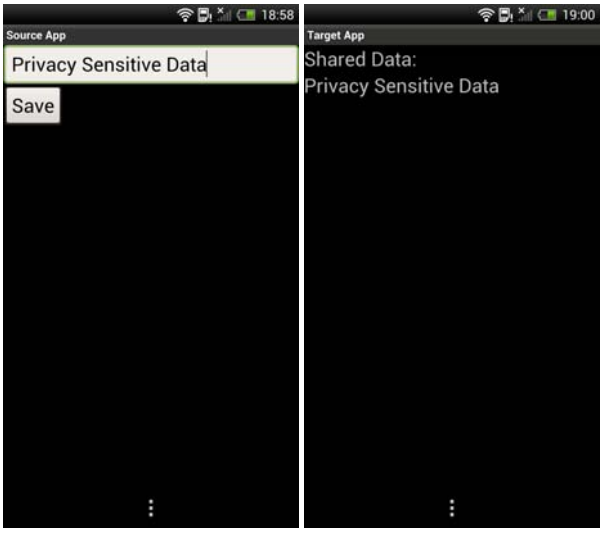
Manuscript received August 5, 2012
Manuscript revised August 20, 2012

```
        (R.id.editText);
      editor.putString("Data", editText.getText().
        toString());
      editor.commit();
    }});  }}
```

Fig. 1 Source code of source application.

```
public class Main extends Activity{
  @Override
  public void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    TextView readData = (TextView)findViewById
      (R.id.readData);
    try{
      Context context = createPackageContext("preference.
        source.app", CONTEXT_IGNORE_SECURITY);
      SharedPreferences pre =
        context.getSharedPreferences
        ("Main", MODE_WORLD_READABLE);
      readData.setText(pre.getString("Data", ""));
    } catch (NameNotFoundException e){
      e.printStackTrace(); } }}
```

Fig. 2 Source code of target application.



(a) Source application. (b) Target application.

Fig. 3 Screenshots of sample applications.

On the other hand, mobile platforms, including Android [3], iOS [4], Windows 8 [22, 23], and Access Linux Platform (ALP) [28, 1] have security enforcement mechanisms based on sandboxes and permissions. Several security analyses of the Android OS have been carried out, and potential attacks were discovered [27, 11, 5, 16]. Enck [14] proposed the Kirin security service for the Android OS, which executes malware detection at the time of installation. Fuchs et al. [25] presented SCanDroid, which statically analyzes the information-flow in Android applications [9] to provide automated security certification. Enck et al. [13] presented TaintDroid, which is an extension to the Android OS and tracks the flow of privacy-sensitive the data in applications. Luo [19] proposed an approach based on static analysis focusing on program slicing. Zhou et al. [33] developed TISSA, which can protect the privacy-sensitive data by sending fake data. Bugiel et al. [5] proposed a security system for Android that monitors application communication channels in the middleware and Linux kernel. Mann and Starostin [20] applied static information-flow analysis to the Dalvik bytecode of Android applications in order to detect privacy violations.

3. Our Approach

We have to grasp a wide range of possible cases of the information leakages to establish a more accurate detection mechanism of privacy breaches. Several schemes have been proposed to improve the security enforcement mechanism of the Android OS by monitoring communications between applications. SCanDroid [25] can track information-flow based on intents that are used for communication between activities of Android applications, and TaintDroid [13] can track intents and file files in an Android device and SD card. However, these schemes cannot monitor all the possible data-sharing methods between applications, and privacy-sensitive data can be leaked via unexpected channels. We show a scenario of an information breach where applications interact using a preference, which is used to store the configuration of an application. The source application that has a privilege to access privacy-sensitive data saves the data into the preference. The target application that has a privilege to use the Internet reads the preference of the source application and displays the data. Figure 1 and Figure 2 show the source code of these sample applications, and the screenshots of the applications are shown in Figure 3.

In the Android OS, only applications with an appropriate privilege can access to privacy-sensitive data or send these data via the Internet. Thus, we can detect most of privacy breaches by checking data sharing between activities in applications. In this paper, we present a detailed manual analysis regarding a wide range of possible methods for sharing data in the Android OS, Data-sharing methods between activities in both the same and distinct applications are considered. Data-sharing methods within the same application are not dangerous themselves; however, we need to monitor them for accurate tracking of privacy-sensitive data. To achieve our goal, we need a technique to check whether privacy breaches are caused

by data-sharing; however, this issue is outside the scope of this paper. We discuss possible solutions in Section 7.

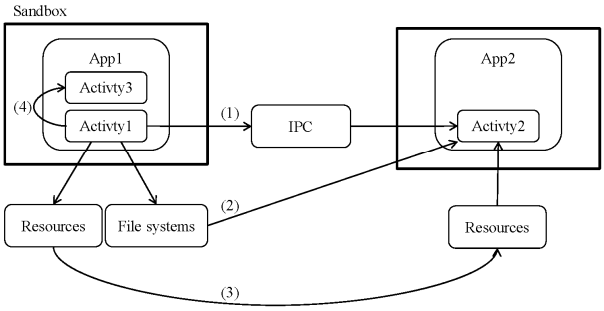


Fig. 4 Data-sharing models.

Table 1: Taxonomy of data-sharing methods in the Android OS

Category	Data-sharing method
(1) IPC	Intent, Remote method
(2) File system	Preferences, SQLite database
	Files
(3) Resources	
	Internet server
(4) IAC	Native code, Static fields, Singleton classes, Application classes

4. Data Sharing

We consider possible data-sharing methods in activity level. An activity is a core component of the Android OS, and each activity represents the task that an Android application can do. A detailed information-flow tracking can be achieved by monitoring data-sharing between activities rather than applications. Thus, we need to check the data-sharing between activities within the same applications as well as distinct applications.

There are four prominent types of data-sharing methods in platforms with a sandbox-based security enforcement mechanism: data sharing based on (1) inter-process communication (IPC), (2) file system, (3) resources, and (4) intra-application communication (IAC). Figure 4 shows these data-sharing models[1].

In the Android OS, ten methods of sharing data are available as shown Table 1. Note that the Android file system is managed and protected by a permission system of the Linux kernel. On the other hand, applications that write to an external storage have to declare an appropriate permission in order to cross the boundary of the sandbox. Thus, the data-sharing method based on files fits into both the file system and resources categories.

4.1 Inter-Process Communication

Data-sharing methods based on intents and remote methods in services fit into the IPC category.

Intent. An intent is an abstract description of an operation to be performed. It is used to launch an activity, send data to any BroadcastReceiver components, and start or bind a service to communicate with a background service.

The source activity generates an intent using the constructor of the android.content.Intent class. Next, the activity sets the package and class name of the target activity using the setClassName() method.

Then, the activity sets the data using the put[Datatype]Extra() method. Then, the activity sends the intent using the startActivity() method of the android.content.Context class to launch the target activity. The target activity obtains the intent using the getIntent() method of the android.content.Intent class.

Then, the activity extracts the data using the get[Datatype]Extra() method.

Remote Method. The Android OS supports remote procedure calls (RPC) between an activity and a service. The interface of a method in the service is described in Android Interface Definition Language (AIDL). An activity can send data to the service, and another activity can receive the data from the service.

The source activity generates an instance of the android.content.ServiceConnection class. Next, the activity generates an intent using the constructor of the android.content.Intent class. Then, it connects to the service using the bindService() method of the android.content.Context class. This method takes three arguments: an intent, the receiver of information returned by the callback methods, and operation options.

We can pass BIND_AUTO_CREATE as the third argument to start the service. Finally, the activity sends data to the service using the data setter method.

The target activity connects to the service in the bindService() method, and receives the data from the service using the data getter method. We assume that appropriate data setter and getter methods are implemented in the service. The interface of these methods should be declared in an AIDL file.

[1] The type 1 method using inter-process message, the type 2 method uses file system managed by the operating system, the type 3 method uses resources protected by the sandbox. The type 1, 2, and 3 methods enable data sharing between activities in both the same and distinct applications. The type 4 method supports only data sharing within the same application; however, it makes difficult to trace the information-flow by combining with other methods.

4.2 File System

Data-sharing methods based on files, preferences, and SQLite databases are in the file system category.

Files. Files in mobile phones and external storage, e.g. an SD card, can be used for data sharing. Activities in distinct applications can exchange data via files in external storage. Note that files under the directory /data/data/[Application name] are used as private storage for each application; thus, only activities within the same application can share data via these files.

The source activity obtains the output stream by the openFileOutput() method of the android.content.Context class. This method takes two arguments: the filename and operating mode. The operation mode should be set to MODE_WORLD_READABLE or MODE_WORLD_WRITEABLE in order to support data sharing between activities of distinct applications. Then, the activity writes the data to the file via the output stream using write() method of the java.io.OutputStream class.

The target activity obtains the input stream by the openFileInput() method of the android.content.Context class. This method takes the file name as an argument. Then, the activity writes the content of the input stream to a buffer with the write() method. Finally, the target activity reads the data in the buffer. Furthermore, the source/target activity can use native code to write/read data to/from the external storage.

Preferences. A preference is used to store persistent data such as configuration information. It stores the pairs of data and key name in an xml file.

A source activity gets an instance of a preference using the getSharedPreferences() method of the android.content.Context class. This method takes two arguments: the preference name and operating mode. The operation mode should be set to MODE_WORLD_READABLE or MODE_WORLD_WRITEABLE in order to support data sharing between activities of distinct applications. Alternatively, the source activity can get the default preference using the getDefaultSharedPreferences() method in the android.preference.PreferenceManager class or the getPreferences() in the android.app.Activity class.
Then, the activity gets the instance of the editor for the preferences using the edit() method of the android.content.SharedPreferences class. Finally, the source activity writes the data to the editor for the preference using the put[Datatype]() methods of the android.content.SharedPreferences.Editor class.

The target activity obtains the context object of the source activity using the createPackageContext() method of the android.content.Context class. Next, the activity obtains the instance of the preference by the getSharedPreferences() method or the getDefaultSharedPreferences() method. Then, the activity reads the data from the preference using the get[Datatype]() method of the android.content.SharedPreferences class.

SQLite Database. The Android OS supports SQLite, which is a lightweight database management system implemented via C library. Activities can share data via the database.

The source activity generates a database using the openOrCreateDatabase() method of the android.database.sqlite.SQLiteDatabase class. This method takes a path to the database file, an optional factory class, and database operating mode. The operation mode should be set to MODE_WORLD_READABLE or MODE_WORLD_WRITEABLE in order to support data sharing between activities of distinct applications. Alternatively, the activity can use the openOrCreateDatabase() method of the android.content.Context class. Then, the activity generates a table by sending the create table SQL query using the execSQL() method. Finally, the source activity adds the data to the database by sending the insert SQL query using the execSQL() method.

The target activity opens the database using the openDatabase() method of the android.database.sqlite.SQLiteDatabase that has the same arguments as openOrCreateDatabase().
Then, the activity receives the data from the database using the query() method.

4.3 Resources

Data-sharing based on files and Internet servers fit into the resources category. The usage of external storage and the Internet are managed by permissions in the Android OS; thus, applications that use the data-sharing methods in this category must declare appropriate permissions. We omit the description of the method using files since it was shown in Section 4.2.

Internet Servers. Activities can exchange data via an external server. We demonstrate data sharing using the HTTP connection. The activity sends the data to a server. Several classes are available to execute the HTTP protocol.

DefaultHttpClient Class
The target activity generates the instance of the org.apache.http.impl.client.DefaultHttpClient class using the constructor. Then, the activity generates the HTTP request containing the data with the constructor of the

org.apache.http.client.methods.HttpGet class. Finally, the activity uses the execute() method of the org.apache.http.impl.client.DefaultHttpClient class to execute the request to the target host.

HttpRequestExecutor Class
The target activity generates the instance of the HTTP request executor using the constructor of the org.apache.http.protocol.HttpRequestExecutor class. Then, the activity generates the HTTP request containing the data with the constructor of the org.apache.http.message.BasicHttpRequest class. Finally, the activity executes the methods preProcess(), execute(), and postProcess() of the org.apache.http.protocol.HttpRequestExecutor class in order to execute the request to the target host.

HttpURLConnection Class
The target activity generates the URL of the external server using the constructor of the java.net.URL class. Next, the activity creates a new connection to the server by the openConnection() method of the java.net.URL class. Then, the activity gets the output stream that connects to the server with the methods getOutputStream() of the java.lang.Process class. The java.io.OutputStreamWriter class is used to convert a character stream into a byte stream. Finally, the activity calls the write() method of the java.io.OutputStreamWriter class to write the data to the output stream.

AndroidHttpClient Class
The target activity obtains the instance of a HTTP client using the newInstance() method of the android.net.http.AndroidHttpClient class. Then, the activity generates the HTTP request containing the data with the constructor of the org.apache.http.client.methods.HttpGet class. Finally, the activity uses the execute() method of the android.net.http.AndroidHttpClient class in order to execute the request to the target host.

DownloadManager Class
The target activity obtains the instance of the DownloadManager using the getSystemService() method of the android.content.Context class. Then, the activity generates the URI with the parse() method of the android.net.Uri class, and the request containing the data with the constructor of the android.app.DownloadManager.Request class. Finally, the activity registers the request to the download manager using enqueue() methods in order to issue the request to the target host.

The target activity receives the data from the server. The activity can use the classes described above.

Furthermore, the source/target activity can use native code to send/receive data to/from the Internet server.

4.4 Inter-Application Communication

Data-sharing methods based on native code, static fields, singleton classes, and application classes are in the intra-application communication category.

Native Code. Android application can call native code via Java Native Interface (JNI). Native code is implemented in a shared object (.so) file. In the native code, activities within the same application can share data using global variables.

The source activity sets the data using the data setter method. The target activity gets the data using the data getter methods. We assume that the application has a native library storing data in a global variable, and the library has appropriate data setter and getter methods.

Static Fields. Fields with a static public modifier can be accessed from any class. Activities within an application can share data using these fields.

The source activity sets the data to static public fields.
The target activity gets the data from the field. We assume that the application has a class for data sharing, and the class has a field with the static public access modifier. The static field can be accessed from any activities within the application in the form of [Class name].[Field name].

Singleton Classes. A singleton class is a class that provides only one instance. Activities within an application can share data via the instance of a singleton class. A singleton class has a static field to store the instance of the class itself. The getter of the singleton instance is implemented as a static method. If the static field is null, the getter generates the singleton instance and returns it. Otherwise, the getter returns the existing instance.

The source activity gets the singleton instance by the instance getter method.
Then, the source activity sets the data by the data setter method. The target activity gets the singleton instance by the instance getter method. Finally, the source activity gets the data by the data getter method.

Application Classes. The android.app.Application class is used as a fundamental class to share the status of the application globally. The activities can share data via the instance of this class. However, the original class has no data setter or getter methods, and it is not convenient for sharing of a large amount of data. Thus, we can extend the class by adding appropriate data setter and getter method.

The name of the extended class and the activity that uses the class should be declared in the manifest file.

The source activity obtains the instance of the class using the getApplication() method of the android.app.Activity class. The activity sets the data to the instance using the data setter methods implemented by the developer.

The target activity gets the instance of the class using the getApplication() method. The activity gets the data from the instance by data getter methods.

5. Exploring the Real World

We studied the data-sharing methods used in actual Android applications. In Google Play [15], applications are classified into 27 categories including games, business, and social. We downloaded 81 applications using Nexus One with Android OS 2.3.6. Then, we selected the top three applications for each category. Table 2 shows the number of applications that has the possibility to use each data-sharing method.

Data sharing based on intents was used in more than 90 percent of the applications. Activities in more than 80 percent of the applications had a possibility to interact with other applications using preferences. The SQLite database and Internet connection were used in about 70 percent of the applications. Many applications potentially use data-sharing methods based on static fields or singleton classes. These methods enable data-sharing with the same application and are not critical by themselves. However, they can make difficult to trace the information-flow by combining with other methods. The rate of applications that have native code was a mere of 13.6 percent. However, data sharing based on native code is difficult to find, and these applications have the potential to become most serious breaches.

Table 2: Data-sharing methods in Android applications

Method	Number of application
Intent	74 (91.4%)
Remote method	35 (43.2%)
File	20 (24.7%)
Preferences	72 (88.9%)
SQLite database	53 (65.4%)
Internet server	57 (70.4%)
Native code	11 (13.6%)
Static fields	71 (87.7%)
Singleton classes	57 (70.4%)
Application classes	17 (21.0%)

6. Check Points for Detection

We describe ten possible data-sharing methods in four categories in Section 4. This section shows how to find the data sharing between activities in Android applications based on Android APIs and access modifiers.

Table 3: Methods for intent manipulation

Type	Method name
1	putExtra(), putExtras()
2	get[Datatype]Extra()
3	startActivity(),startActivityForResult()
4	sendBroadcast(), sendOrderedBroadcast(), sendStickeyBroadcast()

6.1 Inter-Process Communication

Data-sharing methods in the IPC category can be found by the Android APIs for intent manipulation.

Intents. Data sharing based on intents can be found by methods that manipulate intents. There are 38 methods in the classes android.content.Intent, android.content.Context, and android.app.Activity. These methods can be categorized as 1) methods to set the data to an intent, 2) methods to get the data from an intent, 3) methods to launch an activity, and 4) methods to broadcast an intent. Table 3 shows these methods. Note that there are 29 variations of type 2 methods based on the type of target data.

For example, the methods getStringExtra(), getIntArrayExtra(), and getDoubleArrayListExtra() are available.

Remote Methods. An activity launches a service using startService() method in the android.content.Context class. A connection between an activity and service is established by the bindService() method in the same class. Using these methods, we can find data sharing via a remote method in a service.

6.2 File System

Data-sharing methods in the file system category can be found by the Android APIs for file manipulation. We can find the data sharing between activities of different application by checking the operation mode of files. The operation mode should be MODE_WORLD_READABLE and MODE_WORLD_WRITABLE.

Files. Data sharing using the file system can be found using the file manipulation methods. The android.content.Context class has eight methods: getCacheDir(), getDir(), getExternalCacheDir(), getExternalFilesDir(), getFileStreamPath(), getFilesDir(), openFileIutput(), and openFileOutput().

Table 4: Methods for preference manipulation

Type	Method name
1	edit(), apply(), clear(), commit(), remove(), put[Datatype]()
2	contains(), get[Datatype]()
3	getSharedPreferences(), getPreferences(), getDefarultSharedPreferences()
4	registerOnSharedPreferenceChangeListener(), unregisterOnSharedPreferenceChangeListener()

Table 5: Methods for preference manipulation

Method name
execSQL(), getSyncedTables(), delete(), insert(), insertOrThrow(), insertWithOnConflict(), markTableSyncable(), query(), queryWithFactory(), rawQuery(), rawQueryWithFactory(), replace(), replaceOrThrow(), update(), updateWithOnConflict()

Preferences. Data sharing based on preferences can be found using methods that manipulate preferences. Twenty-four methods belong to the classes android.content.SharedPreferences.Editor, android.content.Context, android.preference.PreferenceManager, android.app.Activity, and android.content.SharedPreferences.
These methods can be categorized as 1) methods to set the data to a preference, 2) methods to get the data from a preference, 3) methods to get the instance of a preference, and 4) methods to register or unregister a callback to be invoked when a shared preference is changed. Table 4 shows these methods.

SQLite Database. Data sharing based on the SQLite database can be found by the methods for database operation. The android.database.sqlite.SQLiteDatabase class has 15 methods, as shown in Table 5.

6.3 Resources

Data-sharing methods using the Internet server can be found by the Android APIs for the Internet protocols.

Internet Server. Data sharing via an Internet server can be found with the methods used for internet communication. These methods are shown in Table 6.

Table 6: Methods for internet communication

Class name	Method name
org.apache.http.impl.client. DefaultHttpClient	execute()
org.apache.http.protocol. HttpRequestExecutor	execute()
java.net.URL	openConnection()
java.net.URL	getContent()
java.net.URL	openStream()
android.net.http.AndroidHttpClient	execute()
android.app.DownloadManager	enqueue()

6.4 Intra-Application Communication

Data-sharing methods in the IAC category cannot be found by the Android APIs or access modifiers. The methods based on native code require an information-flow tracking technique or program analysis technique for detection. We have to find field accesses and user-defined setter/getter methods to find methods based on singleton classes.

Native Code. Existence of native code can be found by the declaration in the activity class. However, it is difficult to check whether native code actually shares the data without examining the native library. Thus, we need to use information-flow tracking technique or program analysis technique.

Static Fields. Data sharing based on static fields can be found using access modifiers. Fields with static, public static, protected static, and protected public static are accessible from any class.

Singleton Classes. We can determine that a class is singleton or not by examining the constructor. Data sharing based on singleton classes can be found by access to the fields and methods of the singleton class. We can confirm that a class is singleton by checking the access modifier of the constructor is private.

Application Classes. Data sharing based on an application classes can be found by the getApplication() method in the android.app.Activity class. This method returns the instance of the application class.

7. Consideration

We show how to detect actual privacy breaches in Android applications. One possible technique is to use a data tainting technique. Tainting is a technique that tracks sensitive data. TaintDroid [13] supports message-level tracking, variable-level tracking, method-level tracking and file-level tracking. Privacy breaches using data-sharing methods in the IPC category can be detected by message-level tracing. Breaches using a method in the file system category can be detected by file level tracking. Breaches using data-sharing method based on Internet server can be detected as information-flow against a taint sink. Breaches caused by data-sharing methods in the IAC categories can be detected by variable level tracking. However, TaintDroid modifies the native library loader to ensure that applications can only load native libraries from the firmware. Applications containing native libraries cannot be executed on TaintDroid. The limitation may reduce the flexibility of the development process of Android applications.

The other solution is to check the calls between components of applications. An activity can send privacy-sensitive data to another activity using an intent. In this situation, the activity calls a method in other component or an Android API to obtain the privacy-sensitive data, and the activity calls another Android API to send a intent to the target activity. The run-time enforcement system [2] hooks method/API calls and blocks suspicious calls that

violate a security policy. The security policy is defined as inclusive relation of secrecy labels and integrity labels. If a component access to privacy sensitive data, the data is assigned to the secrecy label of the component. Similarly, if a component has a privilege to manipulate a resource, the privilege is assigned to the integrity label. The call is allowed if and only if caller's secrecy label is a subset of callee's and caller's integrity label is a superset of callee's. Static label information is stored in the manifest file of an application. Dynamic modification of labels is also supported. Floating label enables a caller component to taint its labels to the callee component when the call is allowed. Declassification capability dynamically removes the specific elements in the secrecy label of component and endorsement capability adds the elements to the integrity labels. This system monitors calls between components (activity, service, and component provider), Android or Java APIs, and system calls, and we can check the data-sharing based on inter-process communication, file system, resources, and native code. However, data-sharing based on static fields, singleton classes, and application classes beyond the scope of monitoring. Thus, the detection of these kinds of breaches requires a variable-level tracking technique. One possible solution is to add the variable-level tracking of TaintDroid to the system. Another approach is to rewrite the byte code of the target application [12]. For example, the date-sharing based on static fields, singleton classes, and application classes can be replaced with explicit data-sharing based on intents.

8. Concluding Remarks

In this paper, we showed a wider range of possible methods whereby data can be shared between activities in Android applications, and how to find them. We newly presented inter-application data sharing based on preferences and SQLite database, intra-application data sharing based on native code, static fields, singleton classes, and application classes.

Table 7 shows the assumptions required by each method, category, and check points. Our detection method identifies the points that could potentially cause privacy breaches; however, it is not always true that the identified points lead to malicious data sharing with other activities. Thus, we should use existing framework to find actual privacy breaches as discussed in Section 7. Our results contribute to enhance the security enforcement mechanism in the Android OS.

Table 7: Summary of data-sharing methods

Methods	Assumptions	Categories	Check points
Intent	No	IPC	Android APIs
Remote method	Service	IPC	Android APIs
File	Storage and Permission	File system & Resources	Android APIs
Preference	No	File system	Android APIs
Internet server	Server and Permission	Resources	Android APIs
Native code	No	IAC	Information tracking or program analysis
Static fields	No	IAC	Access modifiers
Singleton classes	No	IAC	Access modifiers and Field references
Application classes	No	IAC	Android APIs

References

[1] Access. Access linux platform security policy framework. http://alp.access-company.com/files/ACCESS_WP_Security-web.pdf.

[2] J. Aljuraidan, E. Fragkaki, L. Bauer, L. Jia, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on android. Technical report, CyLab, Carnegie Mellon University, http://www.cylab.cmu.edu/research/techreports/2012/tr_cylab12015.html, 2012.

[3] Android Developers. Security and permissions. http://developer.android.com/guide/topics/security/security.html.

[4] Apple. Security overview. http://developer.apple.com/library/ios/documentation/Security/Conceptual/Security_Overview/Security_Overview.pdf, 2012.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In Proc. of the 19th Annual Network & Distributed System Security Symposium (NDSS2012), 2012.

[6] A. C. Myers. Jflow: Practical mostly-static information flow control. In Proc. of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'99), pages 228−241, 1999.

[7] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology, 9(4):410−442, 2001.

[8] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In Proc. of the 23rd Annual Computer Security Applications Conference (ACSAC2007), pages 463−475, Dec. 2007.

[9] A. Chaudhuri. Language-based security on android. In Proc. of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS'09), pages 1−7, New York, New York, USA, 2009. ACM Press.

[10] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient _flow tracing with dynamic binary rewriting singapore-mit alliance. In Proc. of the 11th IEEE

International Symposium on Computers and Communications (ISCC2006), pages 7−12, 2006.
[11] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winand. Privilege escalation attacks on android. In Proc. of the International Information Security Conference (ISC2010), Lecture Notes in Computer Science 6531, pages 346_360, 2010.
[12] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A rewriting framework for in-app reference monitors for android applications. In Proc. of Mobile Security Technologies 2012 (MOST2012), 2012.
[13] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. Mcdaniel, and A. N. Sheth. Taintdroid : An information-flow tracking system for realtime privacy monitoring on smartphones. In Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10), 2010.
[14] W. Enck, M. Ongtang, and P. Mcdaniel. On lightweight mobile phone application certification. In Proc. of the 16th ACM Conference on Computer and Communications Security (CCS'09), pages 235−245, 2009.
[15] Google Play. https://play.google.com/.
[16] M. Grace, Y. Zhou, Z. Wang, X. Jiang, and O. Drive. Systematic detection of capability leaks in stock android smartphones. In Proc. of the 19th Annual Network & Distributed System Security Symposium (NDSS2012), 2012.
[17] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In Proc. of the 21st Annual Computer Security Applications Conference (ACSAC2005), pages 303−311, 2005.
[18] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In Proc. of the IEEE International Symposium on Secure Software Engineering (ISSSE2006), pages 87−96, 2006.
[19] K. Luo. Using static analysis on android applications to identify private information leaks. Technical report, Kansas State University, 2011.
[20] C. Mann and S. Artem. A framework for static detection of privacy leaks in android applications. In Proc. of the 27th Symposium on Applied Computing (SAC2012), pages 240−245, 2012.
[21] S. McCamant and M. D. Ernst. Quantitative information-flow tracking for c and related languages. Technical report, Massachusetts Institute of Technology, 2006.
[22] Microsoft. App capability declarations. http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx.
[23] Microsoft. App contracts and extensions. http://msdn.microsoft.com/en-us/library/windows/apps/hh464906.aspx.
[24] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS2005), 2005.
[25] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications. Technical report, University of Maryland, 2009.
[26] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information _ow control. In Proc. of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI'09), pages 63−74, 2009.
[27] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A small but non-negligible flaw in the android permission scheme. In Proc. of the 2010 IEEE International Symposium on Policies for Distributed Systems and Networks (Policy2010), pages 107−110, 2010.
[28] A. Sjöström, K. Fukushima, S. Kiyomoto, W. Shin, and T. Tanaka. Security analysis of access linux platform. IJCSNS International Journal of Computer Science and Network Security, 10(5):12−18, 2010.
[29] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In Proc. of the 4th ACM European conference on ComputerCommunications (EuroSys'09), pages 61−74, 2009.
[30] G. Smith. Principles of secure information flow analysis. Advances in Information Security, 27(5):291−307, 2007.
[31] Symantec. Symantec reports increase in collaborative malware. http://www.internetbusiness.co.uk/2011/03/02/symantec-reports-increase-in-collaborative-malware/
[32] A. Yip, X. Wang, N. Zeldovich, and K. M. Frans. Improving application security with data flow assertions. In Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09), pages 291−304, 2009.
[33] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In Proc. of the 4th International Conference on Trust and Trustworthy Computing (TRUST2011), pages 93−107, 2011.

Kazuhide Fukushima received his M.E. in Information Engineering from Kyushu University, Japan, in 2004. He joined KDDI and has been engaged in the research on digital rights management technologies, including software obfuscation and key-management schemes. He is currently a researcher at the Information Security Lab. of KDDI R&D Laboratories Inc. He received his Doctorate in Engineering from Kyushu University in 2009. He received the IEICE Young Engineer Award in 2012. He is a member of Institute of Electronics, Information and Communication Engineers, the Information Processing Society of Japan, and ACM.

Lujo Bauer is an Assistant Research Professor in CyLab and the Electrical and Computer Engineering Department at Carnegie Mellon University. He received his B.S. in Computer Science from Yale University and his Ph.D., also in Computer Science, from Princeton University. Lujo's research interests span many areas of computer security, and include building usable access-control systems with sound theoretical underpinnings, developing languages and systems for run-time enforcement of security policies on programs, and generally narrowing the gap between a formal model and a practical, usable system.

Limin Jia is a Research Systems Scientist in CyLab at Carnegie Mellon University. She received her B.E. degree in Computer Science and Engineering department at the University of Science and Technology of China. She received her Ph.D. in Computer Science from Princeton University. Her research interests include programming languages, language-based security, logic, and program verification.

Shinsaku Kiyomoto received his B.E. in Engineering Sciences, and his M.E. in Materials Science, from Tsukuba University, Japan, in1998 and 2000, respectively. He joined KDD (now KDDI) and has been engaged in the research on stream cipher, cryptographic protocol, and mobile security .He is currently a senior researcher of the Information Security Lab. in KDDI R&D Laboratories, Inc. He received the Dr. degree in engineering from Kyushu University in 2006.He was a visiting researcher of the Information Security Group, Royal Holloway University of London from 2008 to 2009. He received the Young Engineer Award from IEICE in 2004.He is a member of the Physical Society of Japan and Institute of Electronics, Information and Communication Engineers.

Yutaka Miyake received the B.E. and M.E. degrees of Electrical Engineering from Keio University, Japan, in 1988 and 1990, respectively. He joined KDD (now KDDI) in 1990, and has been engaged in the research on high-speed communication protocol and secure communication system. He received the Dr. degree in engineering from the University of Electro-Communications, Japan, in 2009. He is currently a senior manager of Information Security Laboratory in KDDI R&D Laboratories Inc. He received IPSJ Convention Award in 1995 and the Meritorious Award on Radio of ARIB in 2003.