Detecting Passive Content Leaks and Pollution in Android Applications

Yajin Zhou Xuxian Jiang
Department of Computer Science
North Carolina State University
yajin_zhou@ncsu.edu jiang@cs.ncsu.edu

Abstract

In this paper, we systematically study two vulnerabilities and their presence in existing Android applications (or "apps"). These two vulnerabilities are rooted in an unprotected Android component, i.e., content provider*, inside vulnerable apps. Because of the lack of necessary access control enforcement, affected apps can be exploited to either passively disclose various types of private in-app data or inadvertently manipulate certain security-sensitive in-app settings or configurations that may subsequently cause serious system-wide side effects (e.g., blocking all incoming phone calls or SMS messages). To assess the prevalence of these two vulnerabilities, we analyze* $62,519$ *apps collected in February 2012 from various Android markets. Our results show that among these apps,* $1,279$ *(*2.0%*) and* 871 *(*1.4%*) of them are susceptible to these two vulnerabilities, respectively. In addition, we find that* 435 *(*0.7%*) and* 398 *(*0.6%*) of them are accessible from official Google Play and some of them are extremely popular with more than* $10,000,000$ *installs. The presence of a large number of vulnerable apps in popular Android markets as well as the variety of private data for leaks and manipulation reflect the severity of these two vulnerabilities. To address them, we also explore and examine possible mitigation solutions.*

1 Introduction

Smartphones are becoming increasingly popular. According to a recent report from Gartner [3], the worldwide smartphone sales are increased by 47% in the fourth quarter of 2011. From the same report, Google's Android has already taken over other competing mobile platforms to become the market leader (by having more than 50% of smartphones shipped). Moreover, mobile platform vendors and other relevant parties in the ecosystem (e.g., carriers) create centralized marketplaces or app stores to streamline the distribution of mobile applications (or simply apps). Because of these factors, the total number of apps available for users to download is huge. For instance, as of September 2012, there are around $675,000$ and $700,000$ apps available on Google Play [4] and the Apple App Store [2], respectively.

In this paper, we present a systematic study of two vulnerabilities existing in a large number of Android apps. Both of these vulnerabilities stem from a built-in Android component, i.e., `content provider`, which is by default *accessible* by all running apps on the phone, including untrusted ones.[1] Due to the open access, this particular component could be potentially exploited to not only passively disclose various types of private in-app data, but also inadvertently manipulate certain security-sensitive in-app settings or configurations that may subsequently cause serious system-wide side effects (e.g., blocking all incoming phone calls or SMS messages).

To elaborate, we consider the first vulnerability a `passive content leak` as it can cause affected apps to passively disclose (private) in-app data. This kind of vulnerability is different from previous findings about legitimate apps (e.g., Pandora [9]) as well as in-app advertisement libraries [26] that may *actively* leak private information. Also, we find the passive content leak vulnerability affects a large number of mobile apps in current Android markets, including the official Google Play. In fact, among $62,519$ Android apps we collected from various Android markets in February 2012, 2.0% of them suffer from this vulnerability. These vulnerable apps include popular mobile browsers, widely-used instant messengers (IMs) and social network apps, and even some popular mobile security apps.

Different from the first one, the second vulnerability, i.e., `content pollution`, can be potentially leveraged to further manipulate certain in-app data managed by these vulnerable apps. The manipulated data can be a part of security-sensitive settings (e.g., firewall rules or job lists) that are being enforced or executed by affected apps. As a result, they

[1] With the latest release of Android 4.2 in October 2012, Android apps that target API level 17 will have the default "exported" set to "false" for content providers, hence reducing the attack surface for apps [1]. But problems remain for earlier Android versions and still affect apps that target API level below 17.

may be "influenced" to cause undesirable side effects to the system. Example side effects include allowing or denying certain phone calls and SMS messages from specific phone numbers chosen by attackers. Our study shows that this vulnerability is also widely present in real-world apps. Among the same set of $62,519$ apps, 1.4% of them are affected by this vulnerability.

To systematically assess the prevalence of these vulnerable apps, we have designed and implemented a tool called `ContentScope`. Our tool first examines a given app and checks whether it exposes the public content provider interfaces (represented as `start` functions). If yes, we then locate those Android functions or routines that actually operate on internal databases with private data (denoted as `terminal` functions). After that, our tool performs path-sensitive data-flow analysis along execution paths from `start` functions to `terminal` functions, so that we can automatically derive necessary constraints and prepare "appropriate" inputs to evaluate the presence of content leakage or pollution vulnerabilities. The inputs will then be dynamically fed into the app on a real phone for confirmation.

We have applied the `ContentScope` tool to $62,519$ apps collected in February 2012 from various Android markets, including the official Google Play. Using the tool, we detected $1,456$ vulnerable apps (or 2.3% of our dataset). Specifically, $1,279$ apps (or 2.0%) suffer from `passive content leaks`, 871 apps (or 1.4%) are susceptible to `content pollution`, and 694 apps (or 1.1%) contain both vulnerabilities. Among these vulnerable apps, 435 (0.7%) and 398 (0.6%) of them were downloaded from the official Google Play.

After identifying these vulnerable apps, we further perform a break-down to better understand them. Specifically, by analyzing apps with `passive content leaks`, we aim to understand the types of personal information that may be passively leaked. Our results show that the relevant personal information for leaks includes, but not limited to, (1) incoming and/or outgoing SMS messages (268 apps); (2) contacts stored in the phone (128 apps); (3) conversations in popular IMs such as MSN (121 apps); (4) user credentials such as user name, password and authentication tokens of popular social network websites such as Facebook and Twitter (80 apps); (5) browser history and bookmarks (70 apps); and (6) incoming and/or outgoing phone call logs (61 apps). These vulnerable apps include popular mobile browsers, widely-used instant messengers (IMs) and social network apps, and even mobile security apps.

Similarly, when analyzing apps with the `content pollution` vulnerability, we aim to understand possible side effects from them. Our results show that by polluting internal databases, we could effectively manipulate certain blacklists or whitelists (for outgoing/incoming phone calls as well as SMS messages) that are maintained by vulnerable apps. In other words, security-sensitive settings (such as firewall rules) can be arbitrarily changed by any app on the phone without any permission. Our results further show that some vulnerable apps can be even exploited to download additional unwanted apps (and other arbitrary types of files) from remote servers in the background without user's awareness. (The downloaded app can be later automatically triggered for installation, albeit with user approval.)

Among the detected vulnerable apps, some of them are extremely popular, having been downloaded from Google Play for more than $10,000,000$ times. After identifying them, since February 2012, we have been actively reporting our findings to the corresponding developers. Some developers have taken our reports seriously and immediately followed our suggestions to fix their apps. However, we also experienced some difficulties in communicating with other developers, either by being unable to find their valid contact information or by not receiving any response to our reports.

We note that several recent works [15, 17, 19, 20, 27] also attempted to examine potential risks and implications from unprotected interfaces of Android apps. However, our work differs from them by *not* focusing on invoking escalated privileged operations (e.g., from an app without any permission). Instead, we exclusively focus on the open `content provider` interface of Android apps and study potential risks that may lead to passive privacy leakage and unintended manipulation of security-sensitive data. To the best of our knowledge, our paper is the first to systematically study these two issues and quantitatively report the prevalence of affected apps.

The rest of this paper is organized as follows: In Section 2, we elaborate the threat model and assumptions of this work. We then present the overall system design in Section 3, followed by its implementation details in Section 4. After that, we report the evaluation results in Section 5. We also discuss possible solutions and improvements in Section 6. Finally, we describe related work in Section 7 and conclude the paper in Section 8.

2 Threat Model and Assumptions

In this paper, we assume the following adversary model. In order to launch the above-mentioned passive content leaks and/or pollution attacks, a malicious app needs to be installed on the same smartphone as the vulnerable app. After installation on the phone, we do not assume the malicious app will request any dangerous permission[2] to launch

[2]A dangerous permission means the permission is defined at the dangerous protection level [27] and will need to be explicitly approved by users before being granted to third-party apps. In the following, for simplicity, we will use permission to present dangerous permission if not otherwise specified.

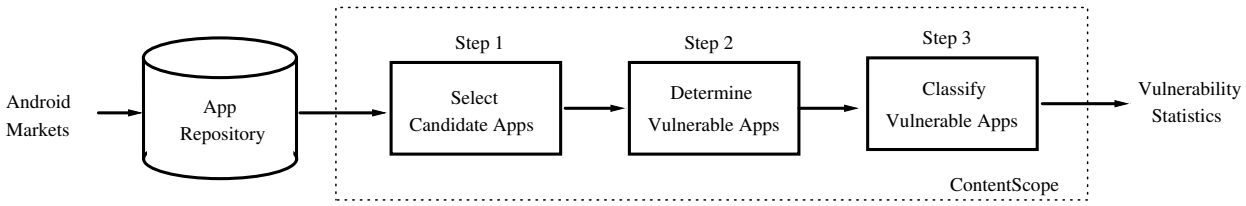


Figure 1. The ContentScope design

the attack. Even so, the malicious app can manage to obtain private or confidential data (such as contacts, SMS messages or browser histories) that may be maintained by vulnerable apps. Further, the malicious app can potentially manipulate certain in-app data that could be a part of security settings or configurations to introduce unintended side-effects in the running system. We note that to transport the stolen private data to a remote server under the attacker's control, the malicious app does need the corresponding INTERNET permission.

3 Design

To analyze these two vulnerabilities and assess their prevalence in existing apps, we design the `ContentScope` system to scalably and accurately examine a large number of apps from existing Android markets. In particular, we consider a passive content leak occurs when an app does not properly protect its internal (private) database and allows it to be freely accessible to any running apps. Similarly, content pollution happens if any app without necessary authorization or permission can manipulate another app's internal database or settings, which can be security-sensitive.

In Figure 1, we show the overall architecture of the `ContentScope` system, which involves several steps to determine vulnerable apps. The first step is to select candidate apps that may potentially be susceptible to these two vulnerabilities. For that purpose, we extract essential vulnerability characteristics to significantly reduce the number of apps that need to be subsequently examined. In other words, the selected candidate apps are expected to be a small portion of all existing apps. After that, the second step is to analyze candidate apps and determine whether they are indeed vulnerable. Finally, to further assess the threat level of these vulnerable apps, we accordingly examine them and present a break-down on the types of leaked or polluted content as well as the associated side effects. In the following, we illustrate each step in detail.

3.1 Candidate App Selection

To select candidate apps, our strategy is to extract essential characteristics of these two vulnerabilities. Specifically, these two vulnerabilities share a similar nature of having an exploitable content provider interface. Note that the content provider interface in Android is designed to concisely encapsulate a structured set of local data (typically in the form of SQLite databases) while providing necessary mechanisms to regulate the accesses to encapsulated data. By default, this interface is *open* so that any app can leverage it to communicate with each other. In the meantime, an app can protect its interface by (1) either setting a property named `exported` to `false` in its manifest file (or more precisely `AndroidManifest.xml`) to ensure that the interface is only available to itself or others with the same user ID or (2) defining custom permissions to expose them only to apps that are granted these custom permissions. Moreover, each (custom) permission has a protection level, which determines how *dangerous* the permission is and the way this permission can be granted to other apps. A certain protection level (i.e., `signatureOrSystem`) is reserved for pre-loaded apps in the phone firmware while others (i.e., `normal`, `dangerous`, and `signature`) can be requested by third-party apps. If there is no protection level specified for a custom permission, the default protection level `normal` will be used.

In this paper, we focus on existing third-party apps available in various Android markets. Therefore, we mainly target custom permissions with `normal`, `dangerous`, and `signature` protection levels. It is also important to note that for a permission at `normal` protection level, any app can request it and the Android runtime will automatically grant access, without asking for user's explicit approval. This differs from permissions at the `dangerous` level, which demand explicit approval from users, or at the `signature` level, which require apps to be signed with the same developer key.

In Figure 2 we show a manifest file which defines a content provider named `ExampleProvider`. Because it does not specify the `exported` property explicitly, this content provider is *open* to all apps on the phone by default. The app defines a custom permission `com.example.app.permission` to protect this content provider. However the `protectionLevel` property of this custom permission is `normal`, which means this permission will be automatically granted to any app that requests it.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/
3            apk/res/android" package="com.example.app">
4      ...
5      <application ...>
6          <provider
7             android:name=".ExampleProvider"
8             android:authorities="com.example.app.provider"
9             android:permission="com.example.app.permission">
10         </provider>
11     </application>
12     <permission
13         android:protectionLevel="normal"
14         android:name="com.example.app.permission">
15     </permission>
16
17 </manifest>
```

Figure 2. An example AndroidManifest.xml file that defines a content provider

Accordingly, to locate those candidate apps, our system first parses their manifest files to determine whether there is any content provider component defined. If yes, we extract the corresponding attributes. The first one is the `exported` property as it specifies whether or not the content provider is accessible by other apps. (If it is not defined, the default `true` value is assumed, which means any app can access it.) The second one is to detect the presence of any custom permission(s) to regulate the read or write accesses to the content provider. More specifically, there are three closely-related ones: `readPermission`, `writePermission`, and `permission`. The `readPermission` and `writePermission` explicitly specify the respective permission used to query and make changes to the data managed by content provider. If they are missing, the `permission` attribute will be used.

In other words, our system chooses those apps as candidates if they explicitly export a content provider by setting `true` in its `exported` attribute or implicitly export this interface without specifying this attribute. In addition, our system also selects those apps, which may have custom permissions, but the corresponding protection level(s) are not defined or defined at the `normal` level. After that, for bookkeeping purposes, we further extract other attributes of the content provider component from these candidate apps, including the `name` property, which specifies the specific class implementing the content provider interface, as well as `authorities`, which is used by the Android runtime to locate the content provider itself. All these relevant attributes will be collected along with the app information and saved into a local database for subsequent analysis.

3.2 Vulnerable App Determination

After selecting candidate apps, our next step is to analyze them to locate vulnerable ones. In order to effectively manage a structured set of data in a local SQLite database, the Android framework provides well-defined APIs to ease the creation and maintenance of content providers. Specifically, these APIs are provided in a number of system-wide classes, including `SQLiteDatabase` (which contains the methods to create, delete and execute SQL commands) and `SQLiteQueryBuilder` (which helps build SQL queries). In other words, content providers can leverage these methods to implement their own standardized APIs that can be invoked to query and make changes to records in local databases. To simplify our discussion, we call these Android APIs that actually manipulate local databases `terminal` functions.

From another perspective, the content provider component essentially encapsulates local content and exports them through standardized APIs. For instance, one standardized API is `query()` that accepts one parameter `URI` to pinpoint which table to query and other parameters to specify query conditions (e.g., the conditions used in the where clause). Another API is `insert()` which handles requests to insert new content into local databases. Consequently, these Android APIs become the actual entry points of content providers to other apps. We call them `start` functions.

Figure 3 shows the implementation of a content provider named `ExampleProvider`. It implements the `query()` (line 17), `insert()` (line 29) and `openFile()` (line 37) interfaces which can be invoked by other apps to operate on the internal SQLite database and internal files. Hence these three functions are `start` functions. In order to manipulate the data maintained by the internal SQLite database, it leverages the methods provided by `SQLiteDatabase` class to perform SQL query (line 25) or insert data (line 33) into database. Hence these two methods are `terminal` functions. Similarly, the function used to open internal files (line 41) is classed as a `terminal` function.

A passive content leak vulnerability exhibits if certain inputs can trigger an execution path from a `start` function to a `terminal` function. Accordingly, we generate a function call graph of a given app to help determine reachability from the public content provider interfaces (i.e., `start` functions) to the low-level database-operating routines (i.e., `terminal` functions). More specifically, we first generate the whole program function call graph for the app and find all the functions containing `terminal` functions. Then for each `start` function, we identify all potential paths from it to corresponding `terminal` functions. Although the construction of whole program function call graph is a well-studied topic, there are certain aspects unique to Android. One example is the resolution of object references, which is the Android counterpart to traditional points-to analysis in binary analysis, as the exact type of a class object needs to be determined before we can actually obtain the list of functions it may invoke. The second one is the call graph discontinuity introduced by the event-driven nature of Android apps (e.g., with extensive use of callbacks or event registrations). As an app may register various callback functions

```
1  public class ExampleProvider extends ContentProvider {
2
3      private static class DatabaseHelper extends SQLiteOpenHelper {
4          public void onCreate(SQLiteDatabase db) {
5              db.execSQL("CREATE TABLE example_table ...");
6              db.execSQL("CREATE TABLE private_table ...");
7          }
8      }
9
10     private static final UriMatcher sUriMatcher;
11     static {
12         sUriMatcher.addURI("com.example.app.provider", "example_table", 1);
13     }
14
15     private DatabaseHelper dbHelper;
16
17     public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
18         if (sUriMatcher.match(uri) != 1)
19             throw new IllegalArgumentException("Unknown URI " + uri);
20         return internalQuery(uri, projection, selection, selectionArgs, sortOrder);
21     }
22
23     private Cursor internalQuery(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
24         SQLiteDatabase db = dbHelper.getReadableDatabase();
25         Cursor c = db.query("example_table", projection, selection, selectionArgs, null, null, sortOrder);
26         return c;
27     }
28
29     public Uri insert(Uri uri, ContentValues initialValues) {
30         if (sUriMatcher.match(uri) != 1)
31             throw new IllegalArgumentException("Unknown URI " + uri);
32         SQLiteDatabase db = dbHelper.getWritableDatabase();
33         long rowId = db.insert("example_table", null, initialValues);
34         ...
35     }
36
37     public ParcelFileDescriptor openFile(Uri uri, String mode) {
38         try {
39             String newPath = uri.getPath();
40             File newFile = new File(newPath);
41             return ParcelFileDescriptor.open(newFile,ParcelFileDescriptor.MODE_READ_ONLY);
42         }
43         ...
44     }
45 }
```

Figure 3. The source code of a content provider implementation

that can be executed at certain events by the Android runtime, these registered callbacks may not show up in the generated function call graph. To solve these problems, we take a conservative approach by annotating these callback registration routines and reconnecting callback routines back to the generated call graph. For example we annotate the methods in class `Thread` and `Handler` and reconnect the function `Thread.run()` to `Thread.start()` and the function `Handler.sendMessage()` to `Handler.handleMessage()`. In Figure 3, we show the possible execution paths from `start` functions to `terminal` functions as dotted lines.

After identifying possible paths from `start` functions to various `terminal` functions, we then need to generate corresponding inputs as evidence to show the feasibility of one particular execution path. To this end, we first generate a control flow graph (CFG) for each function along the path, then leverage data-flow analysis to obtain necessary constraints that may guide the execution path for a specific input. The collected constraints will be fed into a constraint solver to generate the appropriate inputs that satisfy these constraints. Our approach to generate corresponding inputs is summarized in Algorithm 1. Unfortunately, there also exists certain Android-specific aspects that make this process challenging. First of all, content providers heavily use Android-specific APIs to process inputs. For example, an `uri` input of both `query()` and `insert()` functions is processed by `UriMatcher` to return an integer value, which indicates a mapping between the `uri` and the return value. Without the knowledge of this mapping, a constraint solver will not be able to resolve this constraint. For example, the mapping between `uri` and the return value is "content://com.example.app.provider/example_table" to 1 for the content provided shown in Figure 3. Secondly, certain string-related operations in content provider can also introduce problems to derive the associated constraints. Specifically, without understanding the semantics of these string operations, it is hard to generate the constraints for the values involved in these operations. To address that, we choose to model or summarize the operations in Java `String` class and directly take them into account for constraint extraction. Thirdly, in certain cases, we may not be able to model the internals of specific functions. To accommodate them, we conservatively explore all paths if the return values of these functions are used as constraint variables. For the code shown in Figure 3, the generated input for parameter `uri` which can trigger the execution from `start` function `ContentProvider.query()` (line 17) to `terminal` function `SQLiteDatabase.query()` (line 25) is

Algorithm 1: Generating inputs for `start` functions

Input: set of `start` functions, set of `terminal` functions, function call graph
Output: generated inputs for `start` functions

$paths = []$
$starts =$ [`start` function]
$terminals =$ [`terminal` function]
$callGraph =$ [path in function call graph]

foreach $s \in starts$ **do**
  foreach $t \in terminals$ **do**
    if $(s, t) \in callGraph$ **then**
      $paths$.add(s, t);

$constraints = []$

foreach $path \in paths$ **do**
  $constraints$.add(genConstraint(s, t));

$inputs = []$

foreach $constraint \in constraints$ **do**
  $inputs$.add(genInput($constraint$));
return *inputs*

"content://com.example.app.provider/example_table".

Our static analysis approach is conservative and will likely introduce false positives when generating inputs to an execution path. To address that, we further have a dynamic execution module to confirm the calculated inputs. The dynamic execution module is based on a test app that runs in a real Android phone and takes these generated inputs to invoke the exposed content provider interfaces. The test app then records and analyzes the return values to determine the existence of the vulnerabilities. For example if the invocation to `ContentProvider.query()` returns a concrete `Cursor` object, then we can confirm that the content provider interface (or the app) is vulnerable to passive content leaks. Similarly, if the invocation of `ContentProvider.insert()` returns a concrete `URI` object, then the app is susceptible to content pollution.

3.3 Leaked/Polluted Content Break-down

After identifying the set of vulnerable apps, we want to assess the level of threats by classifying the types of leaked content as well as possible side effects caused by the polluted content. For this purpose, we leverage again the earlier function call graph and CFG. More specifically, in order to identify the types of content leaked by an exposed content provider interface, we need to know the types of content that have been saved in the content provider before. Correspondingly, if there is a possible execution path between an Android API that returns certain type of private information (such as contacts) and another API that is used to insert private information into the content provider, then we can know the type of content stored in the content provider. As a result if this app is vulnerable to a passive content leak, we can infer the specific type of private information may be leaked. Similarly, we apply this approach to infer possible sides effects caused by the polluted content.

In Particular, for a given vulnerable app, if there is an execution path from a contact-retrieving API to an API that inserts data into the content provider, we can infer that the type of data stored in content provider could be contacts. Consequently, the type of private content that may be leaked is also contacts. Similarly, if there is an execution path from the `query()` function (of a content provider) to an Android API that blocks incoming SMS messages (`abortBroadcast()`), we may consider this app could block certain SMS messages according to the content or filtering rules injected by attackers.

We stress that the above method will introduce false positives (as the presence of an execution path in the call graph does not guarantee it is actually executed at runtime). As a result, there is still a need to manually verify the findings. Fortunately, the reported execution paths significantly speed up the analysis. Moreover, our prototyping experience also shows that the context information of apps is helpful to understand the semantics or types of information that may be organized in local databases. For example, an instant messenger (IM) app might use the content provider interface to maintain its accounts, buddy list, and conversation logs. In this case, we do not have well-defined Android APIs to infer the relationship between the app and the type of private information saved on the vulnerable content provider. Fortunately we can leverage context information to classify different types of private information.

4 Implementation

We have implemented a `ContentScope` prototype as a mix of Python scripts and Java code. The first two steps in our system, i.e., candidate app selection and vulnerable app determination, were developed using Python with $3,813$ source lines of code (SLOC). The last step, i.e., vulnerable apps classification, was developed using Java, which extends the open source `baksmali` disassembler tool (1.2.6) and introduces additional $2,800$ SLOCs.

To detect the passive content leak vulnerability, our prototype focuses on two different types of `start` functions in content provider that could leak private data: one is standard `ContentProvider.query()` that supports structured data maintained in internal SQLite databases and the other is `ContentProvider.openFile()` that returns a file descriptor to access a file in the app's private data directory. These two entry points, if not protected, can be exploited

by malicious apps to retrieve either arbitrary data stored directly in local SQLite database or any file accessible to the vulnerable app, including those in its private data directory (such as the local SQLite database file itself). For each of them, we then choose the corresponding `terminal` functions that either query the data from internal database or open a file on the phone. The `terminal` functions supported in our current prototype for `ContentProvider.query()` and `ContentProvider.openFile()` are `SQLiteQueryBuilder.query()`, `SQLiteDatabase.query()`, `SQLiteDatabase.rawQuery()` (which are used to submit the SQL query into internal database) and `ParcelFileDescriptor.open()` (which is used to read a file object directly and return the corresponding file descriptor).

After determining the `start` and `terminal` functions, our prototype builds the function call graph (with intra-method control flow graph), extracts the execution paths from `start` to `terminal` functions, derives necessary constraints along those paths and eventually generates corresponding inputs that satisfy these constraints. As an example, the method signature of one `start` function for passive content leak detection is `ContentProvider.query` (`Uri uri`, `String[] projection`, `String selection`, `String[] selectionArgs`, `String sortOrder`). The parameter `uri` is used to determine which type of information needs to be returned or which table in the database needs to be queried. Hence, by enumerating different `uri` values that satisfy the constraints, we can explore different code paths between the `ContentProvider.query()` (`start` function) and other `terminal` functions to retrieve private data maintained in different tables.

Our experience shows that exhaustively exploring all code paths between the `ContentProvider.query()` functions and `terminal` functions may be limited to only query the tables specified in the content provider, not others. In other words, if there exist other tables that are not specified in this content provider, their data may not be queried. To address that, we extend our system to launch SQL injection attacks [8] in an attempt to obtain accesses to those tables. Specifically, `SQLiteDatabase.query()` leverages `SQLiteQueryBuilder.buildQueryString()` to construct the final SQL strings by concatenating several parameters together. For example, for the query function in line 25 of Figure 3, the final query string constructed is "select **projection** from **example_table** where **selection** = **selectionArgs** order by **sortOrder**;". As a result the attackers can pass the *special* `projection` parameter with "$*$ **from private_table;**" to `query()` function so that the final query string will become "select $*$ **from private_table;** from **example_table** where **selection** = **selectionArgs** order by **sortOrder**;". In Android platform, all the SQL statements after the first special character "**;**" will not be executed, so the effective SQL statement executed is "select $*$ **from private_table;**", which essentially returns all the data in table named `private_table`. Note that the SQL injection attack will not work when the content provider explicitly checks the column names in parameters [8] before issuing SQL queries. However, the use of SQL injection extends the reach to other (private) data that may be otherwise missed. In fact, we have identified such apps where private data can only be leaked through SQL injection (Section 5).

Similarly, to detect content pollution vulnerability, we use `start` functions (e.g., `ContentProvider.insert()`, `ContentProvider.update()`) with corresponding `terminal` functions (e.g., `SQLiteDatabase.insert()`, `SQLiteDatabase.insertOrThrow()`, `SQLiteDatabase.insertWithOnConflict()`, `SQLiteDatabase.update()`, and `SQLiteDatabase.updateWithOnConflict()`). Different from the passive content leak case, one additional challenge here is that we need to generate appropriate data that will be injected into the content provider, which requires prior knowledge of the table scheme. In our prototype, we obtain it by inferring the SQL statements used to create table in method `SQLiteOpenHelper.onCreate()`. In many cases, the SQL statements are constant strings or concatenated with constant strings. For example, the SQL statements used to create tables for the content provider in Figure 3 are shown in line 5 and 6. If not, we will then run the apps in an Android emulator with a customized framework with hooks in class `SQLiteOpenHelper` to record the detailed SQL strings used to create the table. Another related challenge is how to choose the right value for each column in the table. The insertion will fail if the chosen value happens to be in conflict with existing ones for the primary key column. In our prototype, we make ten different attempts with each attempt having different random values to minimize the chance of a conflict.

To confirm the detected vulnerabilities, we developed a test app that runs on a real phone. For automation, this app accepts inputs in a configuration file and uses them to invoke each individual content provider interface in the vulnerable app. If the related `ContentProvider.query()` invocation returns a valid `Cursor` object for database access or the `ContentProvider.openFile()` invocation returns a valid `ParcelFileDescriptor`, we will consider this app is indeed vulnerable to passive content leaks. Similarly, if the invocation of `ContentProvider.insert()` returns a new `URI` object, we will mark the app as vulnerable to content pollution. To help the automation of this process, we also develop a shell script to install each potentially vulnerable app, execute our test app (with app-specific configuration file), and retrieve the test results.

Finally, to classify the confirmed vulnerable apps, our current prototype automatically generates the suspect execution paths and aims to classify the type of information saved in local database. Meanwhile, there is a need to

	Total Apps	Candidate Vul. Apps	Vulnerable Apps	
			Passive Content Leak	Content Pollution
#	62,519	3,018	1,279	871

Table 1. Overall detection results in our dataset

fall back to manual efforts for confirmation. Fortunately, these generated execution paths greatly facilitate this process. Our experience shows that it took a single co-author less than three days to classify all 1,456 vulnerable apps. We stress that our manual efforts are only needed to classify the types of private information that might be leaked or polluted. The selection and confirmation of vulnerable apps are still mostly an automated process. Moreover, the intermediate results (such as function call graph and CFG) can be leveraged to greatly reduce the classification overhead.

5 Evaluation

To assess the level of threats from these two vulnerabilities, we have collected 62,519 (free) apps from various Android markets in February 2012. Among these apps, 35,047 were downloaded from the official Google Play and the rest were fetched from ten other popular third-party ones. From these apps, as described in Section 3, our system first identifies apps with exported content provider interfaces as candidates. In our dataset, our system reports 3,018 (or 4.8%) candidate apps (Table 1). The reduction from the initial 62,519 apps to these 3,018 candidate apps is helpful to exclude unrelated ones for processing. Also we find that among 62,519 apps, 4994 of them have content providers and only 1976 (39.6%) of them explicitly protect them either by not exporting them or by declaring *dangerous* permissions. This fact indicates that if one interface is *open* by default, many developers will do so even without realizing this. After that, for each candidate app, our system analyzes it to confirm whether it is indeed vulnerable. In total, our system detects 1,279 and 871 apps that suffer from `passive content leak` and `content pollution` vulnerabilities, respectively. Among these vulnerable apps, 435 and 398 apps were downloaded from Google Play. As mentioned earlier, some of them are popular, with more than 10,000,000 installs from the official Google Play market.

Some apps were not automatically confirmed by our system. Upon manual analysis, we discovered this was due to the following reasons: (1) In some of these apps, the return value of `start` functions such as `ContentProvider.query()` depends on the internal logic of the app. If certain internal logic is not satisfied, the `start` function will simply return an unexpected value, based on which our system will mark it as not vulnerable. For instance, the `ContentProvider.query()` function of `MiTalk Messenger` (version: 2.1.365/365)[3] will check whether there is a registered user account. If not, it will directly return `null` to our test app. Accordingly, we manually add these apps back to the list of vulnerable apps. (2) Certain apps may enforce an access policy in the `start` functions and deny the request from our test app. However, due to improper enforcement, the access policy may be bypassed. One concrete example is the `QQ Browser` (version: 3.0/35), which checks the package name of calling app in its `ContentProvider.query()` function. If the package name of the calling app is in a predefined list, it will honor the request and return a valid `Cursor` object. Note that this access policy blocked our test app's first try. However, after analyzing the app and accordingly changing the package name of our test app, we can still successfully retrieve data from this app. (3) Other apps may also check the signatures of calling apps. In this case, they are not vulnerable as the signature used to sign the app is supposed to be unique and not leaked to others. (4) Some apps may not be properly developed and will essentially throw an exception when running. For example, we find cases where the `authority` attribute specified in manifest file is different from the one used in `URIMatcher`. In this case, the Android runtime fails to find the corresponding content provider. (5) Finally, some apps may use specific Android classes which cannot be returned to another process (or app). For example, some apps return a `CursorWrapper` to `ContentProvider.query()` function. However this object may not be passed to another app (i.e., our test app) and an `UnsupportedOperationException` will be thrown at runtime.

After identifying these vulnerable apps, starting from February 2012, we spent a considerable amount of time on reporting them to the corresponding developers. Some of them fixed the vulnerabilities and released the patched version quickly.[4] Some developers did not respond but fixed the bugs silently. Yet others did not response and chose to completely ignore our report.

5.1 Passive Content Leaks

In our dataset, we detected 1,279 apps that are vulnerable to passive content leaks. In the following, we organize them into several main categories. The overall results are summarized in Table 2. In the table, we show the number of vulnerable apps in each category and the detailed infor-

[3] In this paper, we use versionName and versionCode in its manifest file to uniquely specify an app. For example the versionName and versionCode of *MiTalk Messenger* app is 2.1.365 and 365, respectively.

[4] For example, the developers of *Maxthon Mobile Web Browser* responded within less than one day and released a patched version in two weeks. And the developers of *Match.com - #1 Dating Site* acknowledged our reports and kept us updated about their process of this vulnerability.

Category	# of Apps	Representative Apps (Available on Google Play)			
		App Name	Package Name	Version	# of Installs
SMS messages	268	Message GOWidget	com.gau.go.launcherex.gowidget.smswidget	2.3/17	1,000,000 - 5,000,000
		Pansi SMS	com.pansi.msg	2.06/226	500,000 - 1,000,000
		Youni SMS	com.snda.youni	2.1.0c/67	100,000 - 500,000
		Blovestorm	com.blovestorm	3.2.1/28	100,000 - 500,000
Contacts	128	mOffice - Outlook sync	com.innov8tion.isharesync	3.0/21	100,000 - 500,000
		WaliSMS	cn.com.wali.walisms	3.2.2/39	100,000 - 500,000
		Shady SMS 3.0 PAYG	com.project.memoryerrorthreepayg	1.78/228	50,000 - 100,000
		360 Kouxin	com.qihoo360.kouxin	1.51/96	1,000 - 5,000
Private information in IM apps	121	GO SMS Pro	com.jb.gosms	4.32/69	10,000,000 - 50,000,000
		Messenger WithYou	miyowa.android.microsoft.wlm	2.0.76/2000076	10,000,000 - 50,000,000
		Nimbuzz Messenger	com.nimbuzz	2.0.10/2091	1,000,000 - 5,000,000
		MiTalk Messenger	com.xiaomi.channel	2.1.365/365	100,000 - 500,000
User credentials	80	Youdao Dictionary	com.youdao.dict	2.0.1(2)/2000102	1,000,000 - 5,000,000
		GO FBWidget	com.gau.go.launcherex.gowidget.fbwidget	2.2/15	1,000,000 - 5,000,000
		Netease Weibo	com.netease.wb	1.2.2/12	10,000 - 50,000
		Netease Cloudalbum	com.netease.cloudalbum	Ver 2.2.0/7	5,000 - 1,000
Browser history or bookmarks	70	Dolphin Browser HD	mobi.mgeek.TunnyBrowser	7.3.0/116	10,000,000 - 50,000,000
		Maxthon Android Web Browser	com.mx.browser	2.4.6/2811	500,000 - 1,000,000
		Boat Browser Mini	com.boatgo.browser	3.0.2/1611	500,000 - 1,000,000
		Mobile Security Personal Ed.	com.trendmicro.tmmspersonal	2.1/31	50,000 - 100,000
Call logs	61	Droid Call Filter	com.droiddev.blocker	1.0.23/24	100,000 - 500,000
		Tc Assistant	cn.com.tc.assistant	4.3.0/19	10,000 - 50,000
		Anguanjia	com.anguanjia.safe	2.58/57	10,000 - 50,000
Private information in social network apps	27	GO TwiWidget	com.gau.go.launcherex.gowidget.twitterwidget	2.1/14	1,000,000 - 5,000,000
		Sina Weibo	com.sina.weibo	2.8.1 beta1/154	100,000 - 500,000
		Tencent WBlog	com.tencent.WBlog	v3.3.0/25	10,000 - 50,000

Table 2. Main types of passively-leaked private information and their representative vulnerable apps

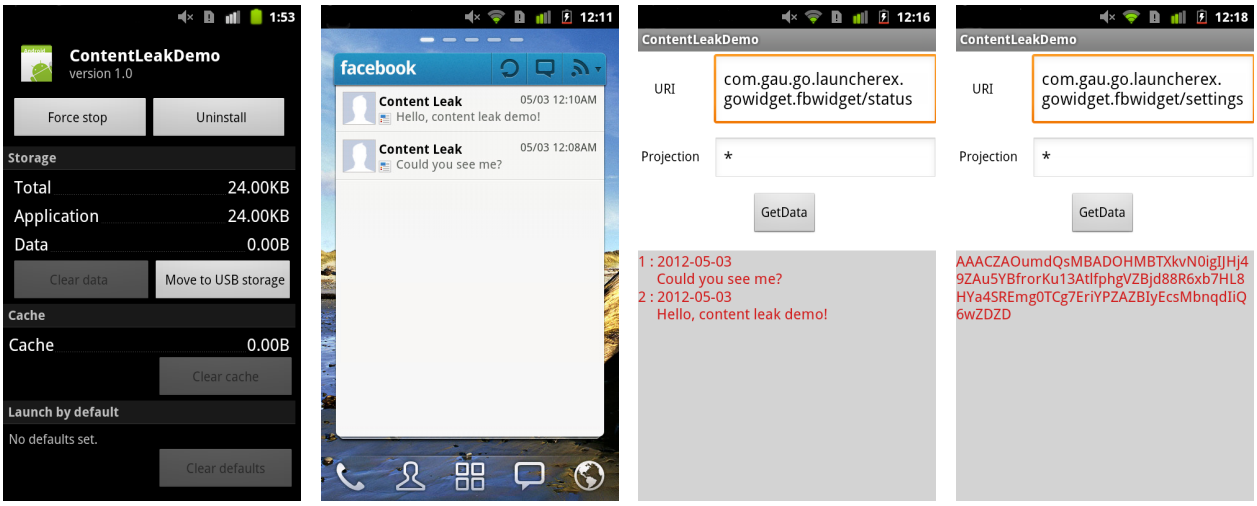
mation of representative apps available on Google Play (including the number of installs – the last column in the table – from Google Play when we confirmed the vulnerability.)

5.1.1 SMS Messages

The first category contains 268 apps that passively leak incoming or outgoing SMS messages stored in the phone. It has been known that sensitive information such as mTANs used in online banking [10] may be present in these SMS messages. Hence, their leaks could pose serious privacy and security threats. As a concrete example, the `Message GOWidget` (version: 2.3/17) app is a widget used in a popular launcher app, i.e., `GO Launcher EX`, which can conveniently display your incoming SMS messages in a desktop widget. Internally, it maintains a database to cache all incoming SMS messages, including their originating addresses and message contents. Unfortunately, this internal database is not properly protected and can be accessed from a public content provider named `.DataProvider`.

`Pansi SMS` (version: 2.06/226) is another example. It is a messaging app that arguably provides more features than the built-in `Messaging` app. By storing both incoming and outgoing SMS messages in the phone and exporting them through an unprotected content provider `.provider.MsgSummaryProvider`, any (untrusted) app in the phone can obtain them, i.e., all SMS messages, without any permission.

In our study, we also notice that some vulnerable apps provide *private SMS message* functionality. With that, mobile users can specify certain phone numbers as *private* so that all SMS messages from these *private* phone numbers



(a) Our test app requests no permission

(b) GO FBWidget displays Facebook posts

(c) Our test app "steals" Facebook posts

(d) Our test app "steals" Facebook AuthToken

Figure 4. Passive content leaks on GO FBWidget

will not show in the built-in `Messaging` app. This feature adds an extra level of privacy for users. Unfortunately, some of these apps define unprotected content providers through which all the *private SMS messages* in the phone can be disclosed to any app! In other words, *private SMS messages* are not *private* any longer but *open* to all apps. One such app is `com.tencent.qqpimsecure` (version: 3.1.1/40).

5.1.2 Contacts

The second category involves private contacts information. One representative app is `mOffice - Outlook sync` (version: 3.0/21), a full-suite productivity app on Android. It can sync contacts, calendars, and tasks with remote desktops. To manage these data for synchronization, it uses an internal database accessible via an open content provider `.dao.DBProvider`. As a result, any app can obtain various types of data saved in this database.

`Shady SMS 3.0 PAYG` (version: 1.78/228) is another example app in this category. It has the feature of configuring certain contacts in the phone as *private contacts*. Similar to in the first category, SMS messages and phone calls from *private contacts* will not show in the built-in `Messaging` or `Phone` app, thus achieving a higher level of privacy. Unfortunately, these `private contacts` in this app are fully leaked via an open content provider `com.project.database.ContactsContentProvider`. Because of a passive content leak vulnerability, using this app does not protect your *private contacts*, but rather put your privacy at risk.

5.1.3 Private Instant Messaging (IM) information

The third category contains personal information in various IM apps. Instant messaging is a convenient form of communication for users and some of them are extremely popular such as Skype, MSN, and ICQ. We found some of these popular IM apps on Android do not protect their data at all. As a result the list of friends in the IM, conversation history, as well as detailed chat messages are all `passively` leaked.

`GO SMS Pro` (version: 4.32/69) is one such instant messaging app with more than $10,000,000$ downloads from Google Play. Besides the enhanced SMS messages functionality, it also provides instant messaging functionality to allow users to communicate with each other. By registering an account (using email address or phone number), users can find their friends and communicate with them on-line. However we find that the friends information and all the conversation content between friends are leaked via a content provider interface, i.e., `ImContentProvider`. Interestingly, in this particular app, we find four content providers and three of them are well protected with the `exported` attribute defined to be `false`. However, the remaining one is not protected, which leaks all internal data.

`Messenger WithYou` (version: 2.0.76/2000076) is an alternative Windows Live Messenger (MSN) app on Android. By using this app, users can chat with their MSN contacts. Similar to the previous app, all the account information and conversation logs are managed in an internal database. Although this app does not have a content provider defined to manage this database, our analysis shows that there exists another unrelated content provider `MiyowaExplorerContentProvider` that imple-

ments an `openFile()` routine. This `openFile()` routine does not check the given file path and thus allows for accessing arbitrary files in the app-specific directory, including the database file.

5.1.4 User Login Credentials

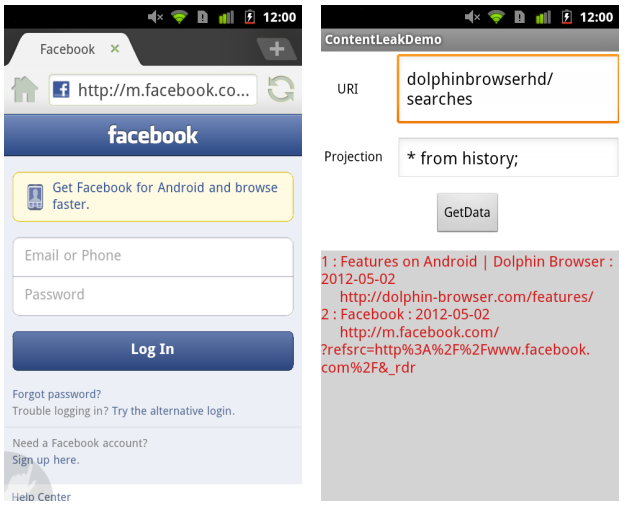
The fourth category includes user login credentials such as username and (plaintext) password of popular websites. These leaked login credentials trivially allow attackers to log into victim's social network accounts. For example, `Youdao Dictionary` (version: 2.0.1(2)/2000102) is a popular dictionary app with more than $1,000,000$ downloads from the Google Play. This app allows users to synchronize their favorite words withe the Cloud using a so-called Netease account. The username and password of this account are saved in plaintext in a local database, which can be leaked via an open content provider. In other words, a malicious app can simply steal the user name and password of the account. It's worth mentioning that this Netease account can also be used to log into user's email service provided by Netease, a NASDAQ company providing service to more than 500 million users [6].

Besides username and password, this category also includes related authentication tokens or simply AuthTokens from popular social network sites (e.g., Facebook). Note the authentication token can be used to automatically sign into a remote website without the inconvenience of typing passwords every time. Hence by stealing the AuthToken, the attackers can effectively sign into victim's social network accounts and steal all the personal information. `GO FBWidget` (version: 2.2/15) is one such app that saves a user's AuthToken of Facebook in a database. Unfortunately this database can be accessed by any untrusted app without any permission. In Figure 4(a), we show a test app that does not request any permission but takes the input from our system to steal the AuthToken from `GO FBWidget`. The leaked AuthToken is shown in Figure 4(d). This particular vulnerable app allows user to make posts to Facebook (Figure 4(b)) and these posts are similarly leaked (Figure 4(c)).

Similar to this particular app, we also identify others: `GO TwiWidget` (version: 2.1/14) leaks Twitter authentication tokens and `Sina Weibo` (version: 2.8.1 beta1/154) passively discloses Sina Weibo authentication tokens [7]. Sina Weibo is a popular blog service with more than 300 million registered users.

5.1.5 Browser History and Bookmarks

The fifth category refers to a common problem in third-party mobile browser apps. They usually implement a content provider to manage browser history and bookmarks. However, the content provider is not properly protected



(a) Dolphin browser visits Facebook (b) Our test app "steals" the browser history

Figure 5. Passive content leaks on Dolphin browser HD version (based on a successful SQL injection attack)

and can be exploited to leak browser history and bookmarks. For example, `Dolphin Browser` (version: 2.2/26) is a popular one available on Google Play. The mini version implements a vulnerable content provider `.bookmarks.BookmarkProvider`, from which the browser history and bookmarks can be retrieved by any app on the phone. The HD version, i.e., `Dolphin Browser HD` (version: 7.3.0/116), similarly leaks the browser history and bookmarks. Unlike the mini version, the HD version does not directly leak the browser history. Instead we need to leverage a SQL injection attack (Section 4) to retrieve it. More specifically, we use "**∗ from history;**" as the `projection` parameter in `query()` method. In this case, we can obtain all the data (including browser history) stored in the table `history`. Figure 5(a) is the screenshot of `Dolphin Browser HD` visiting Facebook website and Figure 5(b) shows the leaked browser history to our test app. `Maxthon Android Web Browser` (version: 2.4.6/2811) is another app that suffers from a similar SQL injection attack to leak the browser history. Other vulnerable mobile browsers include `Circles Web Browser` (version: 0.4.3/18), `ML-Browser` (version: 1.0/1) and `Mchina Browser` (version: 2.6/4).

Interestingly, we also found a *security* app `Mobile Security Personal Ed.` (version: 2.1/31) from TrendMicro that leaks the browser history as well. This security app has a feature to enable safe browsing by detecting potentially malicious URLs. The app collects the visited URLs as well as their risk scores into a local database managed by a vulnerable content provider, which essentially opens

up access to others. Another mobile browser app, i.e., `UC Browser` (version: 7.9.3/43), encrypts the browser history. While such information is still passively leaked, the encryption makes it harder to recover the original browser history.

5.1.6 Call Logs and Others

The sixth category includes those apps with features to manage call logs. For example, `Tc Assistant` (version: 4.3.0/19) aims to be an assistant to users by managing various phone bills. It logs every outgoing call into a local database, which can be queried and unfortunately leaked to any other app. Another one is `Droid Call Filter` (version: 1.0.23/24), which helps users to block unwanted (or harassing) calls and SMS messages. It maintains a black list and blocks the calls from numbers on this black list. All blocked phone calls will be logged into a local database, which can be retrieved by others. Note that leaking call logs from numbers in a black list seems less risky. However, this blacklist can be manipulated via `content pollution` – as shown in Section 5.2.

Besides the above categories, we also found other vulnerable apps such as `GO Email Widget` (version: 1.81/18) and `139Email client` (version: 5.54 /554) leak user's emails; `Match.com - #1 Dating Site` (version: 2.2.0/25) leaks private information about the persons whom the user wants to date with (including related mail address, phone number, location, income, jobs, interests and others) as well as the entire search history on match.com; `Google Music` (version: 4.1.513/513) leaks the songs and artists the users have listened to and `Astrid Task/To-do List` (version: 3.9.2.3/210) leaks the private todo list and other personal notes. When compared with other types of leaked information, the todo list in `Astrid Task/To-do List` may seem less risky. However, the vulnerable app even accepts a raw SQL as input that will be executed in the internal database. As a result, the attackers can inject and execute arbitrary SQL commands such as deleting and updating tables in the databases.

5.2 Content Pollution

In our dataset, we also detected 871 apps that are susceptible to `content pollution` attacks. In the following, we categorize our findings. The first category includes apps that are developed to block SMS and/or phone calls from certain numbers. For example, `DW Contacts & Phone & Dialer` (version: 2.0.9.1-free/198) allows for blocking phone calls based on a blacklist maintained in an internal database. Unfortunately, the interface (e.g., `insert()`) is not protected and allows an attacker to insert arbitrary numbers into the blacklist. In other words, any phone call can be potentially blocked without the user's awareness.

To our surprise, we even found some well-known and popular *security* apps vulnerable to this attack. For example, `qqpimsecure` (version: 3.1.1/45) is a security app with more than $500,000$ installs. The app has similar functionality to block both SMS messages and phone calls from spammers whose numbers are maintained in a database. By polluting the database or its blacklist, any number can be inserted and the app will blindly recognize it as a spam number to block any SMS message or phone calls from it. We point out that blocking certain SMS messages is a common practice [44] in existing malware (e.g., to avoid showing the charged billing information to users). As a result, by launching the content pollution attack, stealthy malware can be created as the questionable behaviors do not exist in malware themselves but with the help of other legitimate or even reputable security apps. Other security apps sharing similar vulnerabilities include `360mobilesafe` (version: 2.2.0/123) and `Anguanjia` (version: 2.58/57).

The second category contains apps that allow for background downloading of apps for installation. Specifically, these apps have the functionality to download apps in the background and then pop up a dialogue asking users to install them. From our analysis, the event to trigger the downloading and installing behavior can be a normal insertion operation into a local database. With these vulnerable apps, such insertion operations can be launched by any (untrusted) app on the phone. As a result, a malicious app can inject a malicious URL into the table so that the malware will be automatically downloaded and then popped up for user's installation. Compared with known infection techniques by mobile malware [44], such installation behavior is more stealthy and can be easily abused to launch an update-attack by automatically downloading and installing malware which is disguised as the updated version of an existing app. Example apps in this category include `Qihoo browser` (version: 1.5.0 Beta/6) and `Baidu Appsearch` (version: 0.8.1 beta/16777516).

6 Discussion

Our study reveals the prevalence of two vulnerabilities in existing Android apps, which motivated us to further examine their root causes and explore possible solutions. As discussed earlier, these two vulnerabilities are rooted in the Android built-in content provider component and developers may fail to fully understand the associated security risks. Specifically, for those content providers that do not explicitly disable the `exported` attribute, earlier Android frameworks (versions before 4.2) by default open them up to any untrusted apps. As a result, if a developer includes a content provider in its app, it implicitly allows others to access – a poor security design.

To fix that, there are two main approaches. From the platform provider's perspective, the default setting

of `exported` attribute should be `false` – so that content provider by default will be exported only to the app itself. From the app developer's perspective, each needs to be aware of potential security risks and properly adopt security mechanisms to protect their content providers. Example security mechanisms include not exporting the content provider or defining custom permissions with `dangerous` or even `signature` protection level. Both approaches have pros and cons. The former requires an over-the-air (OTA) Android platform update on the default setting of the `exported` property, which may introduce compatibility problems for existing apps. (Notice that with the latest release of Android 4.2 in October 2012, content providers by default are no longer exported if developers set `targetSdkVersion` to 17 or higher in the manifest file [1].) The latter does not have the compatibility issue but needs to involve numerous app developers to update their apps to include necessary security checks, which could be a time-consuming process.

In our study, we observe some app developers did attempt to protect their content providers. However the ways they used are not secure and can be readily bypassed. For example, some developers define custom permissions to protect the content provider interface. But the protection level of these custom permissions is `normal`, which means the permission can be granted to any app requesting it (without user's explicit approval). Other developers check the package names of calling apps, which is rather fragile and can be trivially bypassed. One such example is `QQ Browser` (version: 3.0/35), which ensures the calling apps have the pre-defined names, such as "com.tencent.mtt", "com.tencent.qqpim" or "com.tencent.bookmarktest". As shown in Section 5, such protection can be simply circumvented. Meanwhile, we point out that some reputable *security* apps are also vulnerable. The `Mobile Security Personal Ed.` app (version: 2.1/31) developed by TrendMicro suffers from the first vulnerability, i.e., passive content leaks. The `qqpimsecure` app (version: 3.1.1/45), `360mobilesafe` (version: 2.2.0/123) and `Anguanjia` (version: 2.58/57) are susceptible to both attacks, in not only leaking personal SMS text messages and phone call logs, but also blocking text messages and phone calls from specific phone numbers chosen by attackers. Moreover, based on our reporting experience to the developers of vulnerable apps, in a time window of three months only approximately one third of them are actually keen to follow up to issue the patches, which indicates an OTA update may be a better choice.

From another perspective, our current prototype to detect vulnerable apps is still limited and can be improved. For example, we take a conservative way to automatically select candidate vulnerable apps, which may introduce unnecessary false positives. Encouragingly, this selection step still results in a substantial reduction (with only 4.8% remaining apps for subsequent analysis). Also, the lack of context information or internal logic of particular apps could lead to false negatives as well. Specifically, in our current prototype, the dynamic execution module is based on invocation result of `start` functions to determine whether the app is vulnerable. The invocation return may well depend on the internal logic of the app (e.g., the presence of a user account in `MiTalk Messenger`), which can be missed by our prototype. To remedy this, we need to manually confirm them and infer potential side effects. Fortunately, our system automatically generates intermediate results, i.e., inter-method function call graph and intra-method control-flow graph, which greatly speed up our manual analysis. Nevertheless, there is still a need for us to explore innovative ways to overcome these limitations and achieve better automation.

7 Related Work

Smartphone privacy has recently attracted lots of attentions. For example, earlier researches identified worrisome privacy leaks among mobile apps available on both Android [22] and iOS [21] platforms. A few systems have been accordingly proposed to mitigate this problem by revising or extending the framework for better privacy protection. Examples include Apex [34], MockDroid [13], TISSA [46], AppFence [30], Dr. Android [31] and Aurasium [40]. Specifically, they extend the Android framework or repackage the app to provide finer-grained privacy control over an app's access to potentially sensitive information at runtime.

Most recently, research results show that in-app advertisement libraries [26] can also actively leak private information. To mitigate that, AdDroid [35] separates the advertisement functionality from host apps by introducing a new set of advertising APIs and permissions. AdSplit [38] moves the advertisement code into another process. Moreover, mobile malware may also aggressively collect personal information and upload to remote servers [44]. Our work differs from earlier efforts in identifying and quantifying vulnerabilities that allow for passive (instead of active) information leaks.

In addition, another line of research aims to deal with the classic confused-deputy problem or permission leaks [29] on Android. Examples include ComDroid [17] and Woodpecker [27], which employ static analysis to identify such problems in either third-party apps or preloaded apps. QUIRE [20] and Felt *et al.* [25] propose solutions to mitigate them by checking IPC call chains to ensure unauthorized apps cannot invoke privileged operations. Bugiel *et al.* [15] proposes a run time monitor to regulate communications between apps. Our work is similar to them in exposing possible vulnerability present in unprotected An-

droid components. However, our work differs from them by focusing on passively leaking or manipulating internal data managed by apps, *not* invoking privileged operations without permission. A more recent system CHEX [33] takes a static method to detect component hijacking vulnerability that can be exploited to gain unauthorized access to protected or private resources. Although CHEX can be used to *statically* determine potential leak paths from `start` to `terminal` functions, it can not generate the inputs automatically and can not *dynamically* confirm the potentially vulnerable apps. Also our detailed break-down of leaked and polluted content reflects the severity of the problem and practicality of our system.

Beyond each individual mobile app, researchers also aim to measure or study overall security of existing apps in marketplaces. For example, Enck *et al.* [23] studies $1,100$ top free apps to better understand the security and privacy characteristics of existing apps. Felt *et al.* surveys 46 malware samples from three different mobile platforms to analyze their incentives, and discuss possible defenses. Stowaway [24] is proposed to understand over-privileged apps, which request additional permissions beyond their normal functionalities. DroidMOSS [43] and PiggyApp [42] aim to detect repackaged apps in existing mobile app marketplaces. DroidRanger [45] and RiskRanker [28] are two systems that are designed to detect malicious apps in existing Android markets. MalGenome [44] reports a relatively large collection of Android malware and presents various characteristics of them, which will be helpful to guide the development of effective anti-malware solutions. Peng *et al.* [36] use probabilistic generative models to rank the risks of Android apps. Our work is different by primarily focusing on one built-in Android component i.e., content provider, and studying the prevalence of vulnerable apps in current Android markets.

In parallel to the above efforts, researchers are also applying security technologies used on desktops into smartphones (e.g., to achieve better isolation or defense against mobile malware). For example, L4Android [32] and Cells [11] take a virtualization-based approach to isolate different virtual phones. In other words, multiple virtual smartphones can run on one single physical phone side-by-side with necessary isolation. Also related, MoCFI [18] is a framework to enforce control-flow integrity in iOS apps at run time without requiring access to the app's source code. Similar to the systems used for malware analysis on desktop, DroidScope [41] is a system that can be used to analyze Android malware.

To assess the prevalence of vulnerabilities reported in this paper, our system shares a similar spirit with earlier systems that are designed to detect vulnerabilities in desktop software. For example, BitBlaze [39] is a binary analysis framework upon which practical tools can be developed to discover buffer-overflow bugs [37] or detect zero-day exploits [14]. KLEE [16] is a symbolic execution tool that can automatically generate test cases with high coverage. The generated test cases can be used to detect potential bugs existed in programs. AEG [12] is designed to automatically generate exploits for control-flow hijacking attacks. Our system is designed to achieve similar goals for automatically discovering and even generating the inputs (or exploits) to trigger these vulnerabilities. However, our key contributions are not in the tool development itself, but in identifying these two vulnerabilities and measuring their prevalence in existing apps. Moreover, certain differences in the running environments as well as targeted applications between these systems and ours lead to unique considerations in our system design and implementation (Section 3). From another perspective, Java PathFinder [5] is a model checking tool proposed to test Java programs with a custom Java virtual machine. Our system is developed to analyze Dalvik bytecode, which is substantially different from Java bytecode. Nevertheless, these proposed techniques are applicable to enhance our tool for better automation and coverage.

8 Conclusion

In this paper, we present two types of vulnerabilities that are rooted in the unprotected content providers of vulnerable apps. The first one, i.e., `passive content leak`, allows private information managed by a vulnerable app to be passively leaked to any other app without any dangerous permission; the second one, i.e., `content pollution`, allows for unauthorized changes on the internal data managed by vulnerable apps. To assess the extent of these two vulnerabilities, we analyze $62,519$ apps collected in February 2012 from various Android markets. Our results show that among these apps, 1,279 (2.0%) and 871(1.4%) of them are susceptible to these two vulnerabilities, respectively. Also we find that among the vulnerable apps, 435 (0.7%) and 398 (0.6%) of them are downloaded from Google Play. The information being passively leaked ranges from personal contacts, login credentials, call logs, SMS messages, browser histories, etc. Also, the unauthorized manipulation of vulnerable apps' data can be leveraged to block certain phone calls and SMS messages from specific numbers chosen by attackers or download unwanted apps for installation. The presence of a large number of vulnerable apps as well as a variety of private data for leaks and pollution reflect the severity of these two vulnerabilities.

Acknowledgements We would like to thank our shepherd, David Wagner, and the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. We also want to thank Michael Grace, Wu Zhou, Minh Q. Tran, Lei Wu and Kunal Patel for the helpful dis-

cussion. This work was supported in part by the US National Science Foundation (NSF) under Grants 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Android 4.2 APIs. http://developer.android.com/about/versions/android-4.2.html.

[2] App Store (iOS). http://en.wikipedia.org/wiki/App_Store_(iOS).

[3] Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth. http://www.gartner.com/it/page.jsp?id=1924314.

[4] Google Play. http://en.wikipedia.org/wiki/Google_Play.

[5] Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf.

[6] Number of Web users in China Hits 513 Million. http://latimesblogs.latimes.com/technology/2012/01/chinese-web-users-grow-to-513-million.html.

[7] Sina Weibo. http://en.wikipedia.org/wiki/Sina_Weibo.

[8] The Risk You Carry in Your Pocket. https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-slides.pdf.

[9] Your Apps Are Watching You. http://online.wsj.com/article/SB10001424052748704694004576020083703574602.html.

[10] ZeuS-in-the-Mobile - Facts and Theories. http://www.securelist.com/en/analysis/204792194/ZeuS_in_the_Mobile_Facts_and_Theories.

[11] J. Andrus, C. Dall, A. Van't Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, 2011.

[12] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation . In *Proceedings of the 18th Annual Symposium on Network and Distributed System Security*, NDSS, 2011.

[13] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th International Workshop on Mobile Computing System and Applications*, HotMobile, 2011.

[14] D. Brumley, J. Newsome, and D. Song. Sting: an end-to-end self-healing system for defending against internet worms. In *Book chapter in "Malware Detection and Defense", Editors Christodorescu, Jha, Maughn, Song*, 2007.

[15] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS, 2012.

[16] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs . In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2008.

[17] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys, 2011.

[18] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS, 2012.

[19] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security*, ISC, 2010.

[20] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security, 2011.

[21] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Symposium on Network and Distributed System Security*, NDSS, 2011.

[22] W. Enck, P. Gilbert, B.-g. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, USENIX OSDI, 2010.

[23] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security, 2011.

[24] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS, 2011.

[25] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security, 2011.

[26] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec, 2012.

[27] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS, 2012.

[28] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services*, MobiSys, 2012.

[29] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22, October 1998.

[30] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS, 2011.

[31] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: Fine-grained permissions in android applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, CCS-SPSM, 2012.

[32] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, CCS-SPSM, 2011.

[33] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS, 2012.

[34] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2010.

[35] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2012.

[36] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS, 2012.

[37] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2009.

[38] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *Proceedings of the 21th USENIX Security Symposium*, USENIX Security, 2012.

[39] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS, 2008.

[40] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21th USENIX Security Symposium*, USENIX Security, 2012.

[41] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21th USENIX Security Symposium*, USENIX Security, 2012.

[42] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of 'piggybacked' mobile applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, CODASPY, 2013.

[43] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, CODASPY, 2012.

[44] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, IEEE S&P, 2012.

[45] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS, 2012.

[46] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST, 2011.