Android Database Attacks Revisited*

Behnaz Hassanshahi
b.hassanshahi@u.nus.edu
School of Computing
National University of Singapore

Roland H. C. Yap
ryap@comp.nus.edu.sg
School of Computing
National University of Singapore

ABSTRACT

Many Android apps (applications) employ databases for managing sensitive data, thus, security of their databases is a concern. In this paper, we systematically study attacks targeting databases in benign Android apps. In addition to studying database vulnerabilities accessed from content providers, we define and study a new class of database vulnerabilities. We propose an analysis framework to find such vulnerabilities with a proof-of-concept exploit. Our analysis combines static dataflow analysis, symbolic execution with models for handling complex objects such as URIs and dynamic testing. We evaluate our analysis on popular Android apps, successfully finding many database vulnerabilities. Surprisingly, our analyzer finds new ways to exploit previously reported and fixed vulnerabilities. Finally, we propose a fine-grained protection mechanism extending the manifest to protect against database attacks.

Keywords

Android Security; Program Analysis; Database; Malware

1. INTRODUCTION

Many Android apps (applications) use data stored in databases. Furthermore, an app can interact with the database(s) of another app providing functionality to each other. A vulnerable app may allow malware to violate the integrity and confidentiality of data stored in its databases. We call such vulnerabilities *database vulnerabilities*.

We argue that the use of databases in Android apps is significant thus finding and mitigating database vulnerabilities is important. There has been little work on detecting database vulnerabilities in Android. The most relevant work, ContentScope [31], only studies public database vulnerabilities, namely, vulnerabilities arising from content providers. Nevertheless, database implementation in apps is not limited to content providers. Apps can use internal databases without content providers which can be accessed by other means such as Intents. Such databases which we call private databases can also have database vulnerabilities.

In this paper, we study and characterize database vulnerabilities in benign Android apps. In particular, we define the new class of private database vulnerabilities. To detect database vulnerabilities, we propose an analysis framework which automatically generates proof-of-concept exploits. Such exploits can be used by malware on the device to exploit the vulnerabilities. Analysis of database vulnerabilities requires accurate analysis of URI objects which are heavily utilized when using databases, e.g., apps may have buggy validation and sanitization on incoming database access requests using URI-based library methods. We handle this with symbolic models of URI objects whose operations are analyzed with our symbolic execution framework.

Although Android has permissions which deal with public databases, they are not sufficiently fine-grained to prevent public database attacks. It gets worse for private databases as there are no specific permission mechanisms which deal with their usage. We propose a conservative extension to the Android manifest to help developers secure their apps from database attacks.

We have implemented a prototype system, DBDroidScanner, which finds public and private database vulnerabilities and generates the corresponding exploits to confirm them. Using DBDroidScanner, we analyze 924 real-world apps and find a significant number of them having database vulnerabilities. DBDroidScanner automatically generates 153 exploits for those database vulnerabilities. This suggests that database vulnerabilities in Android can be significant. By exploiting these vulnerabilities, unprivileged malware on the device can cause data leaks and pollution. As we generate the malware automatically, it suggests that it may be difficult for anti-virus to detect such malware [6].

We also re-analyze apps whose public database vulnerabilities were studied in [31]. We find that some of these apps still contain vulnerabilities in their updated versions, e.g., `Maxthon Android Web Browser`. More importantly, we found apps where the content provider reported to be vulnerable to the public database attacks in [31] have been updated to fix the vulnerability, turn out to be still exploitable via other components due to private database vulnerabilities. We also find some more complex attacks. In apps which were only studied for public database attacks [31], we find

*This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2015NCR-NCR002-001) and administered by the National Cybersecurity R&D Directorate.

AsiaCCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates.

ACM ISBN 978-1-4503-4944-4/17/04…$15.00.

DOI: http://dx.doi.org/10.1145/3052973.3052994

new privilege escalation attacks which are triggered from private database vulnerabilities but end up exploiting protected content providers (public databases) of other apps.

In summary, our contributions are: (i) a broad classification of database attacks in Android apps; (ii) accurate models for URI-based libraries essential for analysis of apps using databases; (iii) a detection and exploitation framework for zero-day database vulnerabilities; and (iv) a new database protection mechanism for Android apps.

2. PUBLIC & PRIVATE DATABASES

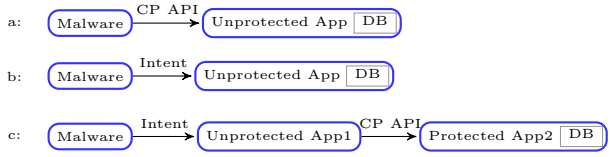

Figure 1: Database Attack Scenarios: a is public; b and c are private attacks. CP API stands for Content Provider API.

Android developers often open up the databases implemented in their apps to "other apps" on the device using content provider components which provide APIs for public database access from other apps. They can also implement databases without exposing them through the public database mechanism by instead using the inter-app communication mechanism, i.e., Intents. In this paper, we call the former group of databases, *public databases* and the latter, *private databases*. The basic idea is that databases in apps are not vulnerable if the APIs which a malware can use are sufficiently protected by permissions (more details of the cases for public and private databases can be found in Sec. 4.1). Thus, a vulnerable app is one whose (public or private) databases are not fully protected. For our purposes, database means what Android provides for data storage, namely the `SQLiteDatabase` library and files (e.g., `ParcelFileDescriptor`).

Public Database Attacks. The first category of attacks targets the public databases which are accessible through content providers. Fig. 1.a shows the public database attack scenario where a malware app uses the parameters of an unprotected content provider API to exploit the database vulnerabilities of a victim app. In Android, components which are meant to be accessible by other apps need to be exported by specifying the `android:exported` attribute in the manifest file. Content provider is a special case. Line 4 in Listing 3 shows the `android:exported` attribute of the content provider tag in the manifest file. By default, this attribute is set to `false` in Android SDK 17 (released Nov 2012) and higher which means that the content provider is not available to other apps. However, content providers in apps built for SDK 16 and lower are exported by default, hence accessible by all apps. In this paper, we study the public database attacks for the SDK 17 and higher.[1] While the `android:exported="false"` attribute isolates a database from other apps, unfortunately, it is coarse-grained preventing all legitimate apps from using public database functionalities.

Developers can protect content provider components using existing or their own custom permissions.[2] Developers can restrict access to the data in content providers across applications at different granularities: (1) setting the `android:exported="false"` attribute is the least fine-grained option to protect the component; Alternatively, they can specify the following permissions: (2) `android:permission` for the whole component which is coarse-grained and prevents apps (malware) lacking this permission from directly accessing the content provider; (3) `readPermission` and `writePermission` which restrict access based on the request, i.e., query or data manipulation. These permissions are more fine-grained and partially protect the content providers; (4) `pathpermission` for protecting particular data stored in databases which is the finest-grained option protecting specific paths of the content providers.[3] In our attack scenarios, we consider cases where the content provider is not fully protected. In Sec. 4.1, we elaborate on what we mean by a content provider which is not fully protected and explain how candidate content providers are chosen.

Private Database Attacks. The second category of attacks targets the private databases which are accessible through inter-app communication. An unprotected app has an unprotected component (except for content provider) which is exported[4] but not protected by any `dangerous` or more restrictive permissions. Fig. 1.b and Fig. 1.c show two private database attack scenarios. In the first scenario, the malware sends malicious Intents to the victim app's components (e.g., activity) to exploit its database vulnerabilities. The second scenario is a privilege escalation attack, e.g., (i) the malware first sends a malicious Intent to the victim App1; next, (ii) App1 invokes the content provider APIs of victim App2 to exploit the vulnerabilities in App2. Note that victim App2 might have correctly protected its content provider with permissions and App1 which is not malware can legitimately have those permissions. It is also possible that victim App1 calls its own protected content provider APIs, in which case only one victim app is involved.

Unlike public databases, Android does not provide any explicit protection mechanism for private databases. Hence, developers have to implement their own (possibly buggy) access-control code to secure internal databases. If a component (e.g., a broadcast receiver) allows an app to access the private databases by sending Intents, there is no further access control mechanism in Android to protect these private databases. Furthermore, many Android apps heavily rely on these private databases to organize various contents such as contacts and app private information.

Intents are the main means of communication in Android. Unfortunately, developers often fail to check its origin properly, e.g. they may use Intents for communication among internal components, but forget that Intents can also be created and sent by other apps. Handling an incoming Intent which modifies the internal database's data is not different from handling other Intents and this might result in the programmer's confusion. As a result, Intents may trigger undesired behaviors which result in private database vulner-

[1] ContentScope [31] analyzes apps for SDK 16 and lower.

[2] A custom permission is declared by the developer and has to be added to the manifest file separately. If the protection level of a permission is `normal`, all applications can get it.

[3] Android allows developers to temporarily override the content provider permissions using the `grantUriPermission`. This case is not considered a vulnerability as the app which owns the content provider should explicitly send an Intent or call `grantUriPermission()` method to allow the app to temporarily access its data.

[4] `exported="true"` is specified or alternatively, the exported attribute is not specified explicitly but the component has an Intent filter.

abilities. Apps may also accidentally expose access paths to private databases by allowing portions of the input string from an Intent to directly be passed to the SQL methods (e.g., `insert()`), which allows attackers to manipulate the database.

The question of whether the developer intends certain functionality allowing other apps (malware) to access its database is a tricky one. With public database attacks, one possible view is that only protecting a specific set of paths of the content provider in the manifest file is intentional. However, a valid counter-argument is that the developer may have chosen a wrong path pattern, thereby creating a vulnerability. We found this to be the case for some of the real app vulnerabilities. Private database attacks are even more susceptible to unintended behavior as the protection in Android is limited only to the permissions specified for the entire exported component which is too coarse-grained. Hence, developers might give up on protecting a component since it restricts the app functionality too much without realizing its exposure to attacks. In this paper, we are conservative and make the reasonable assumption that a database which is available to all apps, including malware, but is not fully protected is a candidate to be analyzed for public and private database vulnerabilities.

3. THREAT MODEL AND EXAMPLES

Threat Model. The adversary in our attack model is malware installed on the Android device. We do not make any assumptions about the permissions requested by the malware, i.e., malware need not request for any permission with `dangerous` or more restrictive protection level. We assume that at least one app on the device is benign but buggy, hence a database vulnerability exists. The malware can attack either public or private databases of unprotected apps as shown in Fig. 1. It needs to craft malicious input (which can be string, data structures, Intent or other object types), sending it to the relevant component of the vulnerable app.

Listing 1 and 2 show two example components of app A (our example benign victim app) which are vulnerable to the public and private database attacks respectively. This example is a simplification of real-world apps. App M in Listing 4 is the malware app which exploits the vulnerabilities in app A.

Vulnerable Public Database Example. Line 4 in Listing 3 shows the content provider tag of app A. The `android:exported` attribute in the manifest allows the content provider to be accessed by other apps on the device. The developer has tried to protect this provider using `<path-permission>` at Line 5. This means that any request which targets URIs with the `"/contacts/"` path (intending to modify the data managed by this provider) will be allowed if the requesting app has the `com.example.app.Write` permission. However, there is a vulnerability in app A (Listing 1). Other paths in this code allow the attacker to pollute the database with sensitive data which will be sent out later (e.g., via SMS).

In the lifecycle of the content providers (see Sec. 4.1), `onCreate()` is the first method called by the Android framework. The two URI patterns are registered in the `android.content.UriMatcher` object: Line 6 maps URIs whose paths only consist of digits to `1` and Line 7 maps URIs whose paths are `"/contacts/"` to `2`. Due to the `<path-permission>` in the manifest file (Line 5 in Listing 3), only the second URI pattern is protected by the `writePermission`. However, since the first URI pattern is not protected by any permission, malware can use it to pollute the database by invoking the `insert()` API in the content provider.

```
1  public class PublicDatabase extends
       ContentProvider {
2    UriMatcher uriMatcher = new UriMatcher(0);
3    PublicDBHelper dbHelper;
4    public boolean onCreate(){
5      dbHelper = new PublicDBHelper(...);
6      uriMatcher.addURI("com.example.app.
         PublicDatabase", "#", 1);
7      uriMatcher.addURI("com.example.app.
         PublicDatabase", "contacts/", 2);
8      return true; }
9    public Uri insert(Uri uri, ContentValues val) {
10     SQLiteDatabase db = dbHelper.
         getWritableDatabase();
11     switch(uriMatcher.match(uri)){
12       case 1:{
13         String table = getTableName(uri.
             getLastPathSegment());
14         if(!table.isEmpty())
15           return db.insert(table, null, val);
16         return null;
17       }
18       case 2:{
19         return db.insert("Contacts", null, val);
20       }
21       default:
22         return null;
23     }
24   }
25   String getTableName(String id){
26     switch(id){
27       case "1": return "ScheduledMessage";
28       default: return "";
29     }
30   }
31   ...
32 }
33 class PublicDBHelper extends SQLiteOpenHelper {
34   public void onCreate(SQLiteDatabase db) {
35     db.execSQL("CREATE TABLE ScheduledMessage (
         message TEXT, ... )");
36     db.execSQL("CREATE TABLE Contacts (_id INTEGER
         , ...)");
37   }
38   ...
39 }
```

Listing 1: Public database example code in app A.

Now, we explain the public database attack launched by app M, the malware in Listing 4. The `insert()` method of the content provider in app A at Line 9 in Listing 1 is called once app M calls `ContentResolver.insert()` at Line 7 in Listing 4. App M also passes appropriate data as arguments of this method to send sensitive messages to the contact list on the device. The `targetUri` at Line 3 is crafted by the attacker in a way to pass the check at Line 11 in Listing 1 to reach the `SQLiteDatabase.insert()` statement at Line 15. In this example, the bug which leads to the public database attacks can be fixed either by enforcing proper permissions in the manifest file or changing the implementation of the content provider. Instead of protecting a particular subset of paths in the manifest file, the developer can protect the whole provider by the `writePermission` but this is inflexible. Alternatively, Line 6 in Listing 1 could be removed to only allow the path patterns that are already protected in the manifest file to execute the code to reach the `insert()` method.

```
1  public class ExampleBroadcastReceiver extends
       BroadcastReceiver{
2    PrivateDBHelper  dbHelper;
3    ...
4    public void onReceive(Context paramContext,
         Intent paramIntent){
5      ContentValues values = new ContentValues();
6      if(paramIntent.hasExtra("task")){
7        String task = paramIntent.getStringExtra("
             task");
8        String data = paramIntent.getStringExtra("
             data");
9        values.put(task, data);
10       if(values.containsKey("message")){
11         int contact =  paramIntent.getIntExtra("
               contact");
12         values.put("contact", String.valueOf(
               contact));
13         handleSendMessage(values);
14       }
15     }
16   }
17   void handleSendMessage(ContentValues values){
18     SQLiteDatabase db =  dbHelper.
           getWritableDatabase();
19     db.insert("SendMessage", null, values);
20   }
21 }
22 class PrivateDBHelper extends SQLiteOpenHelper{
23   public void onCreate(SQLiteDatabase db) {
24     db.execSQL("CREATE TABLE SendMessage (message
           TEXT, ... )");
25   }
26   ...
27 }
```

Listing 2: Private database example code in app A.

Vulnerable Private Database Example. Listing 2 shows an example broadcast receiver in app A which is vulnerable to private database attacks. The manifest file of this app in Listing 3 shows that the broadcast receiver component declares the permission at Line 2 with `"normal"` protection level. Therefore, any application can have this permission including app M. When app M creates and sends a malicious Intent to the broadcast receiver, the `onReceive()` method gets invoked in Listing 2. This method processes the Intent message and if it has a particular pattern (data), parts of its content will be stored in the private database at Line 19.

The private database attack explained above can be prevented by protecting the entry points which make the private internal databases reachable. For example, the broadcast receiver in this example could be protected by a permission with `"dangerous"` protection level. In this way, the malware would not be able to send arbitrary Intents to this component as it does not have the required permission. However, as this protection mechanism is all-or-nothing, it may not be flexible enough. Alternatively, the developer can implement and incorporate validators in the code to prevent the malicious payloads reaching the database. Experience shows that writing a correct validator is error prone, e.g., the vulnerable code in Listing 2 tries to validate the incoming Intent message before inserting its data to the database, but it is still vulnerable.

Our example database attacks show that even though the targets in the private and public attacks are similar, i.e., `SQLiteDatabase`, the attack channels are different and attackers have to bypass different security mechanisms. Public databases have more standardized security mechanisms. Apps with private databases seem to be written more arbitrarily and their security is mostly based on the specific

```
1  <permission android:name="com.example.app.Write"
       android:protectionLevel="dangerous" />
2  <permission android:protectionLevel="normal"
       android:name="com.example.app.permission"/>
3  ...
4  <provider android:name=".PublicDatabase" android:
       authorities="com.example.app.PublicDatabase"
       android:exported="true">
5    <path-permission android:pathPattern="/contacts/
         " android:writePermission="com.example.app.
         Write" />
6  </provider>
7  <receiver android:name="com.example.app.
       ExampleBroadcastReceiver" android:permission="
       com.example.app.permission">
8    <intent-filter>
9      <action android:name="com.example.app.event.
           Trigger" />
10   </intent-filter>
11 </receiver>
```

Listing 3: The content provider and broadcast receiver specification of app A (victim app) specified in its manifest file.

```
1  class DatabaseMalware extends Activity{
2    void insertPublic(){
3      Uri targetUri targetUri = Uri.parse("content
           ://com.example.app.PublicDatabase/1");
4      ContentValues values = new ContentValues();
5      values.put("message","Sensitive Data");
6      ContentResolver contentResolver =
           getContentResolver();
7      Uri confirmationUri =  contentResolver.insert(
           targetUri, values);
8      if(confirmationUri != null){
9        //attack has been launched successfully.
10     }
11   }
12   void insertPrivate(int phoneNo, String message){
13     Intent intent = new Intent("com.example.app.
           event.Trigger");
14     intent.putExtra("task", "message");
15     intent.putExtra("data",message);
16     intent.putExtra("contact",phoneNo);
17     sendBroadcast(intent);
18   }
19 }
```

Listing 4: App M is a malware which accesses the public and private databases of the victim app A.

app implementation. We propose a more flexible mitigation mechanism addressing both public and private database attacks. It can be thought of as adding new private database access controls which are akin to the public ones (see Sec. 5).

The previous examples also show that detecting and generating exploits for database attacks demands accurate analysis of complex objects. A malware installed on the device can construct malicious inputs using data structures and objects which are more complex than primitive types. We also see that there are several libraries with particular semantics for their operations on the execution path. These libraries need to be correctly handled in the analysis to get accuracy. In addition, as the number of entry points for the private database attacks can be large and the private databases can be implemented anywhere in the code-base, the analysis needs to be sufficiently scalable.

4. DETECTION AND EXPLOITATION

Our goal is to detect and exploit the public and private database vulnerabilities in benign apps. At the high level,

finding database vulnerabilities of an Android app can be considered a source to sink reachability analysis problem. The public and private database attacks are triggered by the invocation of the content provider and Intent recipient APIs (e.g., getIntent()) respectively followed by execution paths which reach the sink methods (e.g., `SQLiteDatabase` APIs). In general, static analysis techniques only give potential reachability and existing techniques for Android apps [1, 21, 11] are no different. Moreover, the results of these techniques may give many potential source-sink flows with possibly many false positives [12]. We aim to generate inputs which can exploit these vulnerabilities. ContentScope [31] analyzes public database vulnerabilities using reachability analysis and constraint solving. However, it does not deal with the private database vulnerabilities. Our experience shows that private database vulnerabilities are more scattered throughout the code-base of Android apps. Hence, while more shallow analysis may suffice to find some public vulnerabilities, private vulnerabilities require a deeper analysis with the attendant scalability challenges and neccesity to handle framework libraries more precisely.

We present DBDroidScanner which is based on the analysis proposed in W2AIScanner [12]. In this paper, several components are extended to deal with the analysis and exploitation of database vulnerabilities. W2AIScanner focuses on detecting and exploiting web-to-app injection vulnerabilities. Symbolic execution [5] is used as the core analysis to generate proof-of-concept exploits. In order to balance between precision and efficiency, static information-flow analysis is combined with symbolic execution and dynamic testing. An important difference between DBDroidScanner and W2AIScanner is in the input types supported for exploit generation. The generated exploits in [12] are primitive types (string, integer, etc.). However, DBDroidScanner also needs to support input parameters of non-primitive types which are used by malware to call APIs for exploiting vulnerabilities. Another difference is in the symbolic models used for URI-based libraries. The models utilized by DBDroidScanner support more complex semantics which is necessary to automatically exploit the database vulnerabilities presented in this paper. In this section, we describe our extensions to the analysis in [12].

The analysis framework in [12] employs an initial static dataflow analysis which is fast but imprecise. This is followed by static symbolic execution – a more precise but slower phase. The final dynamic testing of Android apps was found to be the most time consuming phase in practice. Thus, the number of flows generated statically which have to be tested and confirmed by the dynamic testing phase are reduced by having more accurate static analysis.

Similar to [12], DBDroidScanner works in the following phases: (1) it identifies the pairs of source and sink program points and creates the initial CFG (control flow graph); (2) it takes the source-sink pairs produced by the previous step and performs a sink reachability analysis which is utilized as a pre-computation for the search heuristic in the next step; (3) it performs bounded static symbolic execution which results in input constraints which will be employed to generate concrete inputs used in the exploits; (4) the runtime executor constructs the final exploits run on the app to log the execution trace to validate the exploit. In addition, there is feedback from phase 4 to phase 3 to incorporate concrete values obtained from the runtime execution with the path constraints which are solved again by the solver to generate possibly more precise exploits. The static analysis is performed on Jimple, the intermediate representation used in Soot [18]. Next, we discuss the extensions needed for different components of the analyzer to find and exploit public and private database vulnerabilities.

4.1 Source-Sink Pair Identification

The first component of our analysis framework identifies the source-sink pairs. In order to find the entry points which lead to the database sink methods, first, we study possible ways of accessing the public and private databases in the Android apps. We classify sink methods into leakage, pollution and file access categories.[5] The sink methods used in our analysis for the privilege escalation attacks are the `ContentResolver` APIs (e.g., `ContentResolver.insert()`). These sinks are used to detect both public and private database vulnerabilities.

Direct Invocation of Content Provider APIs (Public Database Sources). Content provider components are specially designed to provide shared content among apps. They encapsulate local app data and export them through standardized APIs: `query()`, `insert()`, `openFile()`, etc. These interfaces can be invoked by other apps to operate on the app's internal `SQLite` database and internal files. A malicious app can call these (standardized) APIs to launch pollution, leakage or file access attacks. Once the possible entry methods are determined, we mark the parameters of these methods as source variables.

A content provider is a candidate for analysis if it is not fully protected by appropriate attributes and permissions in the manifest file. A content provider is *fully protected*: (1) by default (equivalent to `exported="false"`) in Android SDK 17 and higher; (2) if `exported="false"` is specified explicitly; (3) if the `android:permission` attribute is specified in the `<provider>` tag in the manifest file.[6] A content provider is partially protected if developers use the `readPermission`, `writePermission` or `<path-permission>` for a more fine-grained protection. If the content provider is protected by these permissions, some of the APIs do not need to be analyzed for the public database attacks. We remove the permission-protected APIs from the source method list based on the category that they belong to and the permission chosen by the developer: (i) if the content provider is protected by the `readPermission`, apps installed on the device must request for it to be able to query the component. Hence, leakage APIs (e.g., `query()`) cannot be a source method; (ii) similarly if a component is protected by `writePermission`, the pollution APIs (e.g., `insert()`) cannot be source methods; (iii) for the APIs which are used to obtain the file handlers (e.g., `openFile()`), `readPermission` and `writePermission` are checked based on the requesting access mode. If the access mode is `"w"`, `writePermission` is checked and if it is `"r"`, the `readPermission` is checked. If the content provider is protected by the `readPermission`, we analyze the app for the `"w"` mode, similarly, `"r"` mode for the `writePermission`. Since the attacker can always gain the file handlers in one of the `"r"`

[5]Some examples are `SQLiteDatabase.query()`, `SQLiteDatabase.insert()` and `ParcelFileDescriptor.open()` for the leakage, pollution and file access categories respectively.

[6]If a content provider is protected by a `permission` which has a protection level higher than `normal` (e.g., `dangerous`), it is not chosen as a candidate entry point for analysis.

or "w" access modes, the file access APIs are always source methods.

A content provider which is protected by the readPermission and writePermission simultaneously will not be chosen as candidate for analysis since such providers are not reachable by the malware. The analysis does not generate an exploit whose reported path is protected by the <path-permission>. For instance, Line 5 in Listing 3 shows that the developer has protected the "/contacts/" path by writePermission. Therefore, our analysis does not generate exploits consisting of URIs which have "/contacts/" path to invoke the insert() method of the content provider as they will be false positives.

Intents and Data Access (Private Database Sources). Android apps may have private databases typically in the form of SQLite databases which are not accessible through content providers. Private databases can be implemented in any component (class) of an app. For private database attacks, we define a component (except for content provider) to be not *fully protected*, if it is not protected by permissions and it is exported (see Sec. 2). Such components are candidates to be analyzed for private database attacks.

A malware installed on the device may indirectly access private databases by crafting malicious Intents without calling any of the methods of the ContentResolver to invoke the ContentProvider APIs. Instead, it sends an Intent that starts a component (e.g., activity), which is part of the provider's app. Hence, the destination component is in charge of retrieving and processing the data. These Intents are obtained by APIs such as getIntent() in activities or onStart() in services, etc., which are our source methods.

4.2 CFG and Reachability Analysis

The CFG construction for analyzing database attacks is similar to the one in [12]. Android apps do not contain a main method, so the CFG is dependent on the lifecycles of the entry point components. The entry point component for the public database attacks is content provider. The content provider APIs are called by the underlying Android framework as callback methods in a specific order as follows. First, the ContentProvider is instantiated and the onCreate() method is called. Next, one of the entry methods of the content provider which is overridden by the app is invoked and this process is repeated for the rest of the overridden entry methods. Notice that the onCreate() method in content providers which is supposed to be called before any other API is anlayzed beforehand. For the entry point components which trigger the private database attacks, we adopt the lifecycles used in FlowDroid [1]. A sink reachability analysis is performed on the CFG to identify whether a sink method is reachable from a program point and its distance from the source.[7]

4.3 Symbolic Execution

While symbolic execution has high precision, it has known challenges [4]. Recently, for the analysis of Android apps using symbolic execution, a more scalable analysis has been proposed [12]. However, challenges remain in static symbolic execution of libraries and frameworks which usually does not scale due to the huge number of paths with long and complex path constraints. Moreover, symbolic execution would then have to deal with low-level data structures in the libraries, e.g., collection classes which are essentially a barrier for many existing analyses. Also, some parts of the libraries might not be supported by the analysis, e.g., native code. Modeling the needed libraries can help us to alleviate these problems.

Modeling Libraries. The analysis introduced in [12] takes a hybrid approach of static symbolic execution and dynamic testing to interact with the Android framework. While the hybrid approach helps for scalability, modeling certain libraries is crucial for certain classes of vulnerabilities. The examples in Listing 1 and 2 show that the execution paths that lead to the database attacks might include Android and Java library methods which have to be handled more accurately by the symbolic execution and constructing exploits to run these paths is not trivial. Among these libraries, those which construct URIs [3] and use its semantics to build filters are particularly important for building public and private database exploits. For instance, the entry methods of content providers which have to be invoked by the malware in the public database attacks require a URI parameter to identify the data in a database. URI objects are different from strings and have more complex semantics. The analysis in [12] supports load and store operations for the fields of URI objects but not more complex operations. As this is not sufficient for generating database exploits, we model the semantics of complex URI methods.

ContentScope [31] which detects public database vulnerabilities does not discuss if it handles such libraries in the analysis. Our approach for handling URI-based libraries combines the classical symbolic execution which is dependent on SMT solvers with automata-based theories. We use the following approach to construct symbolic models for URI-based libraries: (i) we use SMT formulas if a method of a library can be directly translated to an SMT formula and the formula is tractable enough; (ii) sometimes, directly translating the semantics of library methods to SMT formulas is complex and the resulting formula is large (possibly unbounded). If the method maps an input string to an output string, we model them as Symbolic Finite Transducers (SFT) [14] to simulate the I/O relationship. In what follows, we study the structure of URIs and present our models using the approaches discussed above.

Symbolic Representation for URIs. Our analysis keeps a separate pool of URIs to trace the states of the URI instances. We call the model that we have created to represent URIs, *summarized URI*, which conforms to RFC 2396 [3]. A summarized URI object can be altered by the methods of classes such as android.net.Uri.Builder. Basically, our symbolic model for the URI instances follows the original URI class semantics and stores symbolic values for the URI fields. The states of fields of summarized URIs change during symbolic execution. A summarized URI also contains summarized methods which are modeled in one of the following ways: (i) directly translated to SMT formulas; and (ii) SFTs. Summarized URIs are further used as the building blocks of other related classes. For instance, the java.net.UriMatcher class stores the summarized URIs which are added using the UriMatcher.addURI(Uri) method (Line 6 and 7 in Listing 1). A content URI whose scheme is fixed to "content" and identifies data in a content provider is also modeled as a summarized URI.

[7]The distance is used by our search heuristic in the (bounded) symbolic execution phase to choose path traversal to help in reaching sink methods. This helps to make the search more efficient.

Direct Translation of Methods to SMT Formulas. Due to lack of space, we show example symbolic representation of methods of libraries which are dependent on URIs and modeled using individual SMTLib (v2) format formulas. `Uri.getLastPathSegment()` which returns the last path segment of a URI can be modeled with the following self-explanatory constraint:

```
Input: String path
Output: String LPS
Local variable: String x
¬LPS.contains("/") ∧ (path = x."/".LPS ∨ path = LPS)
```

where `uri` is an `android.net.Uri` instance object, `path` is the path field of the `uri`, `x` is a string variable, `LPS` is the output variable for the result of the method which represents the last path segment of the `uri`, `"."` is the concatenation operation and `contains("/")` means that the base variable contains the `"/"` character. We generate the constraint in SMT format to be solved by the SMT solver, for example:

```
(= (str.++ x LPS) path)∧
(= (str.indexof y "/" 0) -1)∧
(or (= (str.++ "/" y) LPS) (and (= (str.len x) 0) (= y
    LPS)))
```

The `getPathSegments()` method is modeled similarly to the bounded version of `String.split` method, i.e., `splitN` where `N` is the maximum number of string tokens that we look for in the base string variable. Some of the methods are translated using models for other methods. For instance, `Uri.Builder.appendId(String id)` is translated using the `Uri.Builder.appendPath(String x)` method. In what follows, we present the symbolic models for library methods which are more complex and therefore modeled using SFTs. We illustrate symbolic execution through concrete examples due to lack of space.

Matching URIs Using SFT. The `UriMatcher.match(Uri)` method is often used in Android programs to compare content URIs. It performs a mapping operation i.e., accepts an object as input and maps it to a unique output value. Modeling this method as an SMT formula is problematic because a symbolic URI may be unbounded. We represent our model using Symbolic Finite Transducers (SFT) [14]. Finite transducers are an extension of finite automata used to model operations on lists of elements. A SFT extends a finite transducer by allowing the transition labels to be predicates.

Fig. 2 shows the SFT designed for the `UriMatcher.match(Uri)` method. In this diagram, U_i belongs to the set of URIs added to an `android.content.UriMatcher` object via `UriMatcher.addURI(Uri)`. Each transition has a constraint (c_i) which outputs `d`, the default value registered in the `UriMatcher` (e.g., `0` registered at Line 2 in Listing 1), if it is not satisfied. φ is the path constraint of the current execution path. If the URI passed as argument to `UriMatcher.match(Uri)` matches any of the U_i, the transducer outputs the integer code stored for U_i (code(U_i)). The symbol *$*, used for the transition between q_2 and q_3 denotes the end of input. Content URI patterns can be matched using wildcard characters. Our symbolic model understands the `"*"` and `"#"` used as the path segment by the `UriMatcher` class where `"#"` is `([0-9]+)` and `"*"` is `(.*)` as regular expressions.

In order to symbolically execute `uriMatcher.match(uri)` at Line 11 in Listing 1, our analysis employs the SFT in Fig. 2

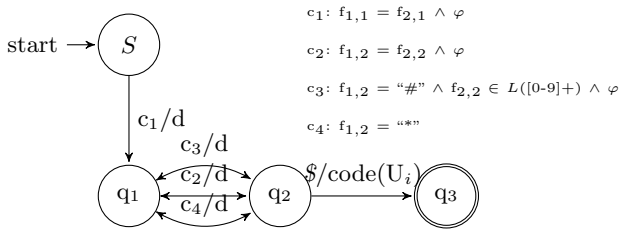

Figure 2: SFT for `UriMatcher.match(Uri)`: φ is the path constraint; the fields of the registered and argument URI are denoted by $f_{1,i}$ and $f_{2,i}$ respectively; c_1 checks the constraints for the authority and c_2, c_3 and c_4 check the constraints for the path segments of the URIs and d is the default integer registered in `UriMatcher` object.

for the two URIs registered in `uriMatcher` at Lines 6 and 7. First the SFT examines whether the argument `uri` matches the URI at Line 6 which restricts the path segment of the `uri` to match `([0-9]+)` and returns `1` (the code registered for the first URI). Next, the analysis examines the second URI at Line 7 which restricts the path to be equal to `"/contacts/"`. However, due to the `<path-permission>` at Line 5 in Listing 3, this URI is protected and should not be reported as exploitable (unsatisfiable due to the path constraint). Hence the path at Line 18 in Listing 1 is infeasible.

Another example method which is modeled using SFT is `Uri.compareTo()`. This method constructs the string representation of the base and argument `Uri` objects and compares them: it returns `0` if the base and argument `Uri` objects are equal; and less or greater than `0` if the base URI string is lexicographically less or greater than the argument URI string respectively. The symbolic model for `Uri.compareTo()` using SFT and an example demonstrating how the model works can be found in Appendix A.

Integrating SFT to DBDroidScanner. One reason for choosing SFT to model the URI-based methods is their compatibility with SMT solvers. This allows us to construct symbolic models whose input are path constraints in the form of SMT formulas and reuse them all over the codebase. The labels of transitions in our implementation for SFT are SMT formulas. At each transition, a new constraint is checked whether it satisfies the existing path constraint. If the constraint is satisfiable and its variables have data dependency on the inputs, it is appended to the path constraint. These constraints can specially help us in generating more precise inputs at the end of the symbolic execution phase. One important characteristic of transducers which makes them useful for modeling URI-based methods is that they can deal with unbounded inputs. This allows us to support URI fields which have recursive structure (e.g., query parameters). The number of iterations for transitioning between states might depend on the loops in the program. In this case, our framework employs a bounded symbolic execution, thereby transitioning for a limited number of times in the transducer. Our implementation for the SFTs is single-valued. Informally, this means that the value returned for a given transition is always a single value. We allow ϵ-transitions in our models by setting the predicate to `"true"` and mapping it to the appropriate value dependent on the states between which it transitions. In this paper, we illustrate SFT models for two example URI methods. Other URI methods (e.g., `Uri.encode(String)`) can also be modeled using SFTs.

Parsing the URIs. Sometimes the analysis needs to construct a URI object for a given URI string (e.g., `Uri.parse (String)` returns a `Uri` object for the `String` argument). URI strings can be parsed using the POSIX regular expression in RFC 2396 to retrieve the scheme, authority, path, query components and fragment parts. In order to model the `Uri.parse(String)` method, first we use the SMT solver to compute a value for the `String` argument which satisfies the path formulas collected so far. Next we use the regular expression to retrieve the fields and construct a URI. If concrete values cannot be resolved for the fields of the URI, symbolic values are generated for them.

4.4 Database Attack Validation

In Sec. 2, we explained how database attacks can be classified into private and public categories. DBDroidScanner extends the validation component in [12] to analyze Android apps for these categories of database vulnerabilities. Our symbolic executor uses an SMT solver (we use the CVC4 SMT solver [20]) to solve the path constraints and generate values for the symbolic input variables identified by the source-sink identification phase. However, generating such values is not adequate for exploiting the vulnerabilities and constructing working exploits using them is not straightforward as shown next. These generated values are processed by the validation component to generate concrete exploits that trigger the private and public database vulnerabilities. We have utilized and designed patterns for generating such exploits based on the source and sink methods of the reported vulnerable path.

Public Database Attacks. Public databases are accessible through content providers. Content providers can be reached from other apps (malware) on the device by directly invoking standardized APIs (e.g., `insert()`). For this purpose, the malware can obtain the content model by calling `getContentResolver()` which allows calling APIs of content providers available to the system. The parameters of these APIs are the symbolic inputs for which the symbolic executor generates values. The validation component uses these generated inputs as well as the manifest file to derive concrete parameters and launch requests to a particular content provider. We explain the content provider exploit generation through an example:

One of the APIs of a content provider is `query(Uri uri, String[] projection, String selection, String[] selArgs, String sortOrder)` which returns a `Cursor` over the result set for the given URI. The `uri` parameter identifies a particular table of a content provider and `projection` is the list of columns to be queried. The `selection` parameter should be formatted as an SQL WHERE clause to enforce constraints on the query and if it contains `?s`, they will be replaced by the `selArgs` parameter. Finally, the `sortOrder` determines how to order the rows in the result set. The symbolic executor in the analysis framework generates values for each of the method parameters. The `uri` parameter starts with the `content` scheme which is fixed in content URIs. The authority segment is obtained from the manifest file and the symbolic executor checks whether its value is consistent with the authority value resolved from the analysis of the program. The reason is that sometimes developers make mistakes and apply conditions on the execution path which prevent the authority registered in the manifest file to be accepted by the content provider. In this case, the content provider cannot handle any request from other apps. Our analysis avoids reporting such false positives.

The remaining segments of the `uri` identify the tables of a database which are generated by the symbolic executor. Some of the parameters of the `query` API have array types. In practice, reasoning about arrays symbolically is not trivial and does not scale well. We only focus on the arrays which are the source method parameters and keep track of their elements in a separate pool. For the rest of the arrays, we do not distinguish their elements and only propagate dataflow facts for the base array object.

It is also possible to set all of the parameters of the `query` API except the `uri` to null. If `projection` is null, all of the columns and if the `selection` is null, all of the rows for the given URI will be returned. If sortOrder is null, results will return with the default sort order.

In order to perform the dynamic testing, we have created a malware skeleton app for invoking the APIs of vulnerable content providers with malicious arguments which are resolved from the static analysis. Our malware does not have any permission granted from the user. Once the vulnerable content provider is invoked, the validator component logs the concrete execution trace using the Android debugger to obtain the concrete values. Using these concrete values, our analysis attempts to place them in the path constraints generated by the static symbolic execution to create more precise inputs if needed. If the generated inputs do not change anymore, our malware invokes the vulnerable content provider and validates the vulnerability using the following sample rules. We assume the `openFile()` and `openAssetFile()` APIs of a content provider are exploited if they return non-null `ParcelFileDescriptor` and `AssetFileDescriptor` references respectively. Similarly, the `query()` API of a content provider should return a non-null `Cursor` reference; the `insert()` API of a content provider should return a non-null `Uri` reference; and the `update()` API of a content provider should return a non-zero integer.

Private Database Attacks. The difference between private and public databases is that public databases are accessed through content providers, while private databases are accessed via Intent messages received by any of the following components: activities, services or broadcast receivers. An input string obtained from an Intent which triggers paths down to the `SQLiteDatabase` methods may allow attackers to manipulate the database and compromise the security of the app. In order to trigger and validate the private database attacks, the attacker should generate Intents which target the vulnerable component of the victim app. For this purpose, the values generated by the symbolic executor are embedded in an Intent message in the validation phase to construct an Intent exploit. A malware can send explicit malicious Intents to a particular component of an app by explicitly setting the target class name using the `Intent.setClass()` API. Alternatively, it can construct an Intent which conforms to the Intent filter of the target component as shown at Line 13 in Listing 4. The validation component collects information about the entry component by parsing the manifest file and the results from the static symbolic execution. It creates data parameters which match the Intent filter and satisfy the path constraints in symbolic execution. In Intent filters, path is one of the data elements that is checked by the Android runtime for accepting an Intent. The developer can specify a special form of regu-

lar expressions as the path pattern. Some of these values might also be obtained from the symbolic execution phase in which case we directly use the values generated by our symbolic executor.

Intent messages transmit data in the following ways: (i) a data URI which references the data resources consists of the scheme, host and path as well as query parameters which are the key-value mappings preceded by the "?"; (ii) Intent extras, the key-value pairs whose type can also be specified in the Intent (e.g, int, string, etc); (iii) other Intent parameters such as categories, actions, etc., that can be sent as string values. An Intent can be constructed as a Java object from a malware app as discussed in this paper. It can also be represented as an Intent hyperlink and invoked from web as explained in [12]. One difference between Intent objects and Intent hyperlinks is that Intent objects can contain arrays and parcelable `extra` parameters while Intent hyperlinks can only contain primitive type `extra` parameters. Hence, it is also possible for the attackers to send malicious data through parcelable key-value pairs and the victim receives the malicious inputs by invoking `Intent.getParcelableExtra()`. We partially support this API for the parcelable types which have been modeled by our system (e.g., Intent). For this purpose, analysis should first resolve the type of the parcelable `extra` parameters received in the target app. For example, if a cast operation is applied to the parcelable parameter, the cast type will be used as the resolved type for the parameter. If the resolved type is supported by our analyzer, an object will be instantiated in the malware program and set as an `extra` parameter to the Intent object. Once the analysis framework generates the key-value pairs as explained in [12] and other necessary inputs for the source-sink flows and the Intent filter specifications for the target component, all of these elements are put together to generate an Intent message.

In order to perform dynamic testing, we configure our malware to send out an Intent message with malicious parameters. Once the target component gets invoked and receives the Intent, the validator logs the execution trace to obtain the concrete values. Similar to the public vulnerability validation, our analysis attempts to place these concrete values in the path constraints generated by the static symbolic execution to create more precise inputs if possible. If the sink method is reached on the execution path and the malicious Intent parameters are observed in the sink stack frame, our validator confirms that the vulnerability is exploitable.

5. MITIGATING DATABASE ATTACKS

We propose a protection mechanism to mitigate both public and private database attacks. Although Android already provides a fine-grained security mechanism for public database attacks, here, we propose an even finer-grained manifest specification. In order to protect public databases, Android allows developers to protect the data referenced by a particular path (pattern) accessible by a content provider. Such path patterns are already enforced at runtime by Android. We propose a generalization of the path patterns in the manifest specification for a content provider. For private database attacks, the entry point components can be protected by an *all-or-nothing* approach but this is too inflexible. We propose a new association between Intents and Android permissions which can make the use of Intents closer to that of content providers and the public database protection specifications.

Our analysis framework can be used to automatically generate the new proposed extensions to the manifest simply by adding the path patterns or Intent permissions discussed below to prevent the vulnerability leading to the generated exploit. Also, sometimes the manifest file does not contain any permissions for URI paths which lead to the public database attacks but the code-base contains execution paths which can be triggered and controlled by attackers. In such cases, our analysis is able to identify the unprotected paths for which appropriate path patterns can be added to the manifest file. While the developer is best placed to make use of our mitigations, since only the manifest is involved, the user (security analyst) can also add the additional specifications to the manifest, repackage and sign the APK file. The red lines in Listing 5 show how the manifest file of the vulnerable app in Listing 3 is extended to prevent public and private database attacks.

Our mitigation is a conservative extension to the Android manifest specification, making use of the existing runtime mechanisms. So, no changes to the code-base of the vulnerable app are needed. The advantage of our proposed mitigation is that it integrates with our analysis to automatically mitigate public and private database attacks. Our analysis can generate proof-of-concept input parameters as part of Intents and URIs which can be blacklisted in the manifest file automatically to prevent the detected attacks.

Mitigating Public Database Attacks. We have observed that Android permissions are limited to specific path patterns supported by the existing implementation of `android:pathPattern` which neither accepts regular expressions nor the syntax supported by URI-based libraries such as `UriMatcher`. Hence, supporting special path patterns used in such libraries is not provided by Android. For instance, the developer can register a URI with `"#"` as a path in the `UriMatcher` to accept a URI whose path pattern is `([0-9]+)`. To address this mismatch, we propose to allow such patterns in the path patterns in the manifest. For this purpose, the implementation of the `android:pathPattern` in the framework needs to be extended. We can then leverage our analysis framework to find the path patterns which need to be protected. Naturally, the developer can also manually determine such fine-grained path patterns. Next, we add these patterns with the appropriate permissions to the manifest file. For the vulnerable public database in Listing 1, our analysis finds that the URIs that have `"#"` path pattern are vulnerable to pollution attacks. Therefore, we add a new tag, `<path-permission>` for the `"#"` path pattern which is protected by `writePermission` (Line 5 in Listing 5).

Mitigating Private Database Attacks. We propose an extension to the manifest specification which allows us to protect execution paths in the program that can be triggered with a particular set of Intents. Since this does not exist in Android, we extend the existing Android manifest specification to be able to protect execution paths by permissions when a particular part of the incoming Intent follows a specific pattern. Such patterns can be obtained automatically using our analysis framework.

The extension decomposes Intents into sub-parts which have primitive types and enforces the access control on Intents matching the specified patterns. These parts include: (1) the string representation of the data URI which includes

```
1  <permission android:name="com.example.app.Write"
       android:protectionLevel="dangerous" />
2  <permission android:name="com.example.app.Read"
       android:protectionLevel="dangerous" />
3  ...
4  <provider android:name=".PublicDatabase" android:
       authorities="com.example.app.PublicDatabase"
       android:exported="true">
5    <path-permission android:pathPattern="#" android:
         writePermission="com.example.app.Write" />
6    <path-permission android:pathPattern="/contacts/"
         android:writePermission="com.example.app.Write"
          />
7  </provider>
8  <receiver android:name="com.example.app.
       ExampleBroadcastReceiver" android:permission="com
       .example.app.permission">
9    <intent-filter>
10     <action android:name="com.example.app.event.
           Trigger" />
11     <intent-permission android:writePermission="com.
           example.app.Write">
12       <extra-pattern android:key="task" android:value
             ="message"/>
13       <extra-pattern android:key="data"/>
14       <extra-pattern android:key="contact"/>
15     </intent-permission>
16   </intent-filter>
17 </receiver>
```

Listing 5: We modify the manifest file of app A in Listing 3 with our extension to protect against database attacks. The extended specifications are colored in red.

the scheme, host, port, path and query parameters; (2) the action; (3) the Intent type; (4) the Intent category; (5) the primitive extra parameters, etc. We extend the `<intent-filter>` tag in the manifest with a new sub-tag, `<intent-permission>` (similar to `<path-permission>` for content providers), e.g., Line 11 in Listing 5. This new extension allows us to protect apps against specific incoming Intents with regular Android permissions which are chosen based on the vulnerable sink method.[8] The `<intent-permission>` can have its own sub-tags to identify specific content patterns. The system checks the permission of the requesting app if the incoming Intent matches the patterns in all sub-tags of the `<intent-permission>`. Listing 5 shows how the specification for the broadcast receiver of the vulnerable app in Listing 2 is extended to protect against private database attacks. We add a new tag, `<intent-permission>` which protects the component with `writePermission` against the set of Intents matching the `<extra-pattern>` tags which refer to the extra parameter of Intents. These tags could also be `<uri-pattern>` and `<query-pattern>`. The `<uri-pattern>` can have attributes such as `android:scheme`. The `<query-pattern>` and `<extra-pattern>` can have `android:key` and `android:value` attributes which can be determined by our analysis framework. For this particular example, the Intents can be matched using `<extra-pattern>` tags.

6. EVALUATION

We analyze real-world Android apps to detect and exploit public and private database attack vulnerabilities. Our main goal is not only to detect potential vulnerabilities but also to confirm them with successful zero-day exploits. We analyze 924 apps (the APK files) in total which belong to the top

[8] E.g., `writePermission` for the `SQLiteDatabase.insert()` sink method.

Table 1: Overall statistics of apps vulnerable to the database attacks.

Category	Sub-Category	# of Vulnerable Apps
Public Databases	Pollution	19
	Leakage	27
	File Access	26
Private Databases	Pollution	12
	Leakage	14
	File Access	5

100 apps across all categories in Google Play and candidate apps analyzed in [31]. Of these apps, 133 apps have at least one exposed and unprotected content provider and all 924 apps have at least one exposed and unprotected component other than content providers.

We ran DBDroidScanner in Ubuntu 12.04 on an Intel Core i5-4570 (3.20GHz) with 16GB of RAM. On average, the static part of our analysis takes from tens to hundreds of seconds. We found analyzing public vulnerabilities tends to be faster than private, as private analysis is more complex due to symbolic execution needing to traverse more and longer paths. To validate the database exploits, our custom malware app is configured to launch the components and perform the privileged operations (e.g., inserting data into the app's database). Listing 4 shows a code fragment of the malware app used to perform public and private database attacks. The runtime execution of a single dynamic test varies depending on the app from seconds to 1 minute. Although DBDroidScanner is a prototype, we see that the times are usable. The static analysis phase is comparable or faster than dynamic execution.

6.1 Database Vulnerability Detection Results

We ran our analyzer on 924 apps where 133 apps have unprotected content providers in the manifest. Hence, we analyze 133 apps for public database attacks and all 924 apps for potential private database attacks which can be exploited via inter-app communication. As shown in Table 1, we detect and confirm 52 public and 23 private vulnerable apps and 153 vulnerabilities in total. We also classified our results based on the content leakage, pollution and file access categories. More detailed case studies of representative vulnerable apps in our dataset are given in Appendix B.

Our results show that modeling the URI-based libraries are necessary to generate accurate exploits for both public and private database attacks. Even though the mechanisms through which the private database attacks are launched (Intents) are different from the public database attacks (content provider APIs), sometimes similar constraints are used by the developers to validate the incoming input. In Appendix B, we discuss two example apps to explain why a good model of such libraries is needed for exploit generation.

6.2 Comparison with ContentScope

We also compare our system with ContentScope [31], the closest related work. As their data set is not available, we have tried to collect and analyze representative vulnerable apps mentioned in the paper. Some of these apps are removed or updated – the original versions are no longer available. Where possible, we analyze these apps under their lower SDK assumption (Android 16 and below) to able to compare with their results. Table 2 shows our results for the representative apps analyzed by ContentScope which are still available.

For the available apps with the same version, our tool is able to find all public database vulnerabilities reported by ContentScope. Additionally, we report private database vulnerabilities in these apps (ContentScope is not designed to find them) and show that 5 of these apps are vulnerable to both public and private database attacks. There are also updated versions of apps in which we do not find public database vulnerabilities but are vulnerable to the private database attacks. We detect and confirm 8 apps vulnerable to the private database vulnerabilities. Surprisingly, there are some cases (`mOffice - Outlook sync`, `Dolphin Browser HD`, `Shady SMS 4.0 PAYG`) where the private database vulnerability allows the attacker to access the protected content providers and launch privilege escalation attacks. In some cases (`Pansi SMS`, `mOffice - Outlook sync`), even though the vulnerabilities reported by ContentScope are no longer applicable in the updated versions of the apps, we find new public database vulnerabilities. We present a more detailed comparison of our results with [31] for public attacks in Appendix B.1, then present our new private database attacks for those apps.

In summary, in these apps which are originally analyzed for public database attacks, we found all three combinations of public and private attacks to be possible (public and private alone, and also combined). Furthermore, combining public and private vulnerabilities can mean more sophisticated and complex attacks.

Table 2: Database vulnerabilities in representative apps of [31]: [31] only deals with public databases, the private column is new. Column three shows the closest APK version we could find. It is marked with "+" if has been updated since the publication of [31]. Column four and five show the minimum and target SDK for which an app is compiled respectively. The last two columns show our findings for the public and private database attacks. In the public column, "✓" means there is still a public database vulnerability, while blank means it is fixed. In the Private column, if we find a private vulnerability we use "✓" (not considered by [31]) and blank otherwise. We use "-" if the dynamic testing could not be done due to some runtime issue (e.g., requires registering a valid phone number.)

ID	App Name	Version	SDK_M	SDK_T	Public	Private
1	`Pansi SMS`	3.6.0 +	7	13	✓	✓
2	`Youni SMS`	4.6.7 +	8	11		
3	`mOffice - Outlook sync`	3.7.7 +	5	-	✓	✓
4	`Shady SMS 4.0 PAYG`	3.38 +	8	18		✓
5	`360 Kouxin`	1.5.0	5	-	✓	-
6	`GO SMS Pro`	7.0.3 +	14	22	✓	-
7	`Messenger WithYou`	2.0.90 +	4	-	✓	✓
8	`Nimbuzz Messenger`	4.1.0 +	15	21		-
9	`MiTalk Messenger`	7.3.32 +	14	19		-
10	`Youdao Dictionary`	6.5.1 +	11	19		
11	`GO FBWidget`	2.2	5	-	✓	✓
12	`Netease Weibo`	2.4.0 +	8	10		✓
13	`Dolphin Browser HD`	11.5.3 +	14	17		✓
14	`Maxthon Android Web Browser`	4.5.8. 2000 +	8	21	✓	
15	`Boat Browser Mini`	3.0.2	7	7	✓	-
16	`Mobile Security Personal Ed.`	7.0 +	9	23		
17	`Droid Call Filter`	1.0.23	4	-	✓	
18	`Tc Assistant`	4.5.0 +	7	-	✓	
19	`GO TwiWidget`	2.1	5	-	✓	✓
20	`Sina Weibo`	6.3.1 +	14	23		
21	`Tencent WBlog`	6.1.2 +	10	-		

7. RELATED WORK

In the past few years, many different aspects of Android security have been studied. Even though many works aim to detect vulnerabilities in benign apps [11, 7, 13, 1, 21], few works also try to exploit them. Database vulnerabilities which may compromise the security of system has been partially studied in [31, 21].

ContentScope [31] is the first work on public database vulnerabilities for Android apps. It finds public database vulnerabilities in Android apps built for the Android SDK 16 or lower in which the content provider components are by default accessible by other apps on the phone. Our work considers the new class of private database vulnerabilities and the corresponding attacks. Another difference is that we focus on apps whose SDK target is 17 and higher where the default assumption has been changed to reduce the occurence of public database attacks. We only analyze the representative apps in [31] which have lower SDK targets under their lower SDK assumption to compare our results with ContentScope. We also propose a framework for analysis of public and private database vulnerabilities with the key being modelling of URI semantics and other Android libraries to help symbolic execution.

CHEX [21] is an information flow analysis tool which is tailored to detect component hijacking vulnerabilities in benign Android apps. Component hijacking attacks happen when an unauthorized app, issuing requests to one or more exported components in a vulnerable app, seeks to read or write sensitive data. CHEX reports the potential vulnerabilities pertaining to the private databases. Compared to this work, our analysis system takes one step further and generates working exploits for the detected vulnerabilities.

Privacy leakage and privilege escalation [9] attacks are two more broad classes of attacks that have attracted researchers during the past few years. Privacy leakage might happen as the result of over-privileged malware or privilege escalation attacks due to application bugs or flaws in the system design. A few systems have been proposed to mitigate these classes of attacks [24, 10, 2, 22, 15, 30, 28, 16]. Where the patching and repackaging works [30, 28, 16] are the closest to our mitigation approach. Jinseong et al. [16] try to improve the coarse-grained policies by banning particular data values to reach a security critical method. However, the analysis used to determine such data values might not be precise enough. Our proposed approach utilizes precise analysis such as symbolic execution and only modifies the manifest file to provide more fine-grained policy enforcement while [16] needs to modify the source code of the app.

Android apps often use libraries such as `android.net.Uri` to perform operations on structured data. Octeau et al. [23] propose a constant propagation framework to resolve fields of URI and Intent objects at a given program point. However, our system synthesizes URIs and Intents which drive the execution of the program along a particular path. Moreover, it is not clear how they handle important operations such as `UriMatcher.match(Uri)` or `Uri.compareTo(Uri)`, modeled using SFTs in our work.

There is a body of work which support string manipulation operations statically for different applications [25, 8, 17, 26]. We transform the utility methods of URI classes to SFTs [14, 27] and leverage the CVC4 SMT solver [20]. Another line of research [8, 19, 29] determines the values of string expressions at a given program point. Our approach

differs from these works as we intend to generate strings that drive execution paths to reach a program point. Java String Analyzer [8] models flow graphs of string operations to a context-free grammar which is over-approximated to a finite state automaton. Instead, we use the existing SMT solvers which support both numeric and string constraints.

8. CONCLUSION

In this paper, we study the database attacks targeting Android apps and propose an analyzer which can find and confirm database vulnerabilities in apps. We show that many popular Android apps from the Google Play store have public and private database vulnerabilities and confirm them by generating exploits. Even apps which have been previously reported as having public database vulnerabilities, can still be vulnerable or exploited in new ways. We also propose an extension for Android to mitigate the public and private database attacks. As our extension only requires modification to the manifest file, it can protect apps without having to fix the vulnerable app code.

9. REFERENCES

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, 2014.

[2] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. Styp-Rekowsky. AppGuard – Enforcing User Requirements on Android Apps. In *TACAS*, 2013.

[3] T. Berners-Lee, R. Fielding, U. Irvine, and L. Masinter. Uniform Resource Identifiers (URI), 8 1998. RFC 2396.

[4] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS*, 2008.

[5] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *USENIX*, 2008.

[6] Z. Cai and R. H. C. Yap:. Inferring the detection logic and evaluating the effectiveness of android anti-virus apps. In *CODASPY*, 2016.

[7] P. P. Chan, L. C. Hui, and S. M. Yiu. DroidChecker: Analyzing Android Applications for Capability Leak. In *WiSec*, 2012.

[8] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, 2003.

[9] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *ISC*, 2010.

[10] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX*, 2011.

[11] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012.

[12] B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena, and Z. Liang. Web-to-Application Injection Attacks on Android: Characterization and Detection. In *ESORICS*, 2015.

[13] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *SAC*, 2013.

[14] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *USENIX*, 2011.

[15] 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 *CCS*, 2011.

[16] 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 *SPSM*, 2012.

[17] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A Solver for String Constraints. In *ISSTA*, 2009.

[18] P. Lam, E. Bodden, L. Hendren, and T. U. Darmstadt. The soot framework for java program analysis: a retrospective. In *CETUS*, 2011.

[19] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond. String Analysis for Java and Android Applications. In *ESEC/FSE*, 2015.

[20] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *CAV*, 2014.

[21] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *CCS*, 2012.

[22] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *ASIACCS*, 2010.

[23] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *ICSE*, 2015.

[24] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *ACSAC*, 2009.

[25] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *S&P*, 2010.

[26] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *CCS*, 2014.

[27] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic Finite State Transducers: Algorithms and Applications. In *CIAA*, 2012.

[28] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX*, 2012.

[29] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic String Verification: An Automata-Based Approach. In *MCS*, 2008.

[30] M. Zhang and H. Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *NDSS*, 2014.

[31] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *NDSS*, 2013.

APPENDIX

A. OTHER SYMBOLIC MODELS FOR URI-BASED LIBRARY METHODS

Fig. 3 depicts our model for `Uri.compareTo(Uri)` as an over-approximation of this method: it returns `0` if the string representations of the two URIs are equal (i.e., transitions reach the accepting state q_7) and `1` otherwise. If the fields of the base `Uri` and the argument `Uri` are represented by $f_{1,i}$ and $f_{2,i}$ respectively and φ denotes the path constraint computed so far, c_i which is the constraint label of a transition is: $f_{1,i} = f_{2,i} \wedge \varphi$. For instance, the transition from the start state, S, to q_1 symbolically represents all possible scheme fields in the base URI ($f_{1,1}$) which match the scheme field of the argument URI ($f_{2,1}$) and also satisfy the path constraint.

We explain how our symbolic model for `Uri.compareTo()` works using an example. If we add the two methods in Listing 6 to the `PublicDatabase` class in Listing 1 which is vulnerable to the public database pollution attacks, this class will also become vulnerable to the public database leakage attacks. In order to generate working exploits, our analysis needs to compare the two `Uri` instances, `u1` and `u2` at Line 4 in Listing 6. Our symbolic executor first creates a summarized URI instance for `u2`, the first argument of the `query` method which is a `Uri` object. Next, it analyzes `supportedUri()` at Line 2. This method builds and returns a new `Uri` object, hence the symbolic executor creates another instance of a summarized URI and initializes its fields with the corresponding values (e.g., `scheme` field will be `"content"`). Line 3 enforces constraints on the fields of `u2` as well and adds them to the path constraint. At this point, the path constraint for this part of the program[9] is:

ϕ = $f_{2,1}$.contains("content") $\wedge$ $f_{2,2}$ = "com.example.app.PublicDatabase" $\wedge$
$f_{2,3}$ = "/".x_1."/".x_2 $\wedge$ $\neg$ x_1.contains "/" $\wedge$ $\neg$ x_2.contains "/" $\wedge$ $x_1 \neq$ null

where x_i is the (i-1)th path segment determined by the `Uri.getPathSegments()` at Line 3 and the fields of `u2` are denoted by $f_{2,i}$: $f_{2,1}$ is the scheme, $f_{2,2}$ is the authority and $f_{2,3}$ is the path field of `u2`. The authority `"com.example.app.PublicDatabase"` is obtained from the manifest file.

At each transition of the SFT, new constraints are concatenated to the path constraint as shown below and if they are satisfiable, the transducer moves to the next state:

c_1: $f_{1,1}$ = $f_{2,1}$ $\wedge$ $f_{1,1}$ = "content"
c_4: $f_{1,2}$ = $f_{2,2}$ $\wedge$ $f_{1,2}$ = "com.example.app.PublicDatabase"
c_6: $f_{1,3}$ = $f_{2,3}$ $\wedge$ $f_{1,3}$ = "/profile/1"
c_7: $f_{1,4}$ = $f_{2,4}$ $\wedge$ $f_{1,4}$ = "id=1"
c_7: $f_{1,5}$ = $f_{2,5}$ $\wedge$ $f_{1,5}$ = "type=admin"

where the fields of `u1` and `u2` are denoted by $f_{1,i}$ and $f_{2,i}$ respectively. The indices in this example are from 1 to 5 and refer to the scheme, authority, path, first query parameter and second query parameter respectively. If the analysis doesn't find any constraint for a transition, it moves to the next state. Since the constraints for all transitions in this example are satisfied, the SFT returns zero which means that `u1` and `u2` are equal. For example, c_1 enforces the $f_{1,1}$ (scheme) field of `u1` to be equal to both `"content"` and $f_{2,1}$ and the path constraint also enforces the $f_{2,1}$ field of `u2` to contain `"content"`. The concatenation of the path constraint and c_1 is satisfiable and restricts the $f_{2,1}$ field of `u2` to be equal to `"content"`. Similarly, the constraints for the rest of transitions are satisfiable and as a result, the sink method at Line 6 can be reached by the malware on the device. The transducer for `Uri.compareTo()` deals with the multiple query parameters in the URI instance (Line 13) using c_7. In this case, the SFT iterates through the query parameters stored in the summarized URI and moves to the accepting state if all the constraints are satisfiable. Note that the query parameter pairs in URIs are implemented using Java container classes. In order to obtain the constraint for a pair, we keep track of individual loaded and stored elements during symbolic execution.

[9]We do not present the path constraint for the previously analyzed parts of the program for simplicity.

```
1  public Cursor query(Uri u2, String[] projection,
       String selection, String[] selectionArgs, String
       sortOrder) {
2    Uri u1 = supportedUri();
3    if(u2.getScheme().contains("content") && (u2.
         getPathSegments().get(0) != null)){
4      if(u1.compareTo(u2) == 0){
5        SQLiteDatabase db = dbHelper.getReadableDatabase
             ();
6        db.query(u2.getPathSegments().get(0), projection
             , selection, selectionArgs, null, null,
             sortOrder, null);
7      }
8    }
9    return null;
10 }
11 Uri supportedUri() {
12   Builder builder = new Uri.Builder();
13   builder.scheme("content").authority("com.example.app
         .PublicDatabase").appendPath("profile").
         appendPath("1").appendQueryParameter("id", "1")
         .appendQueryParameter("type", "admin");
14   Uri u = builder.build();
15   return u;
16 }
```

Listing 6: Example Android program which calls the `Uri.compareTo(Uri)` method.

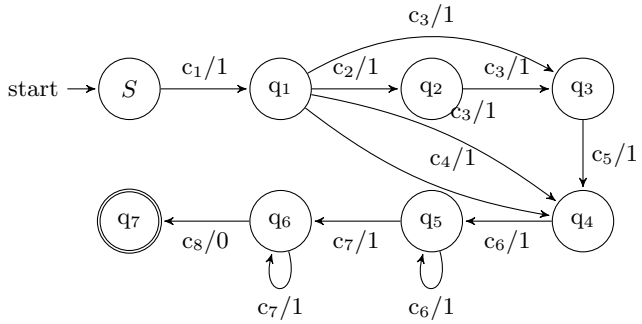

Figure 3: SFT for `Uri.compareTo(Uri)`. The label for each transition is a constraint (c_i) for a particular field of the base and argument URIs: c_1 for scheme, c_2 for userinfo, c_3 for host, c_4 for authority, c_5 for port, c_6 for path, c_7 for query pairs, c_8 for fragment.

B. CASE STUDIES FOR REPRESENTATIVE VULNERABLE APPLICATIONS

We now discuss some findings for the representative vulnerable apps from our dataset. The public and private vulnerabilities are further categorized into pollution, leakage and file access attacks.

`Chomp SMS` (version 6.07) is an SMS app which provides sev-

eral features such as sending scheduled SMS. This app is vulnerable to the pollution and leakage public attacks. The exported content provider, `provider.ChompProvider`, allows the malware to steal or manipulate sensitive data such as scheduled and MMS messages. `CallApp - Caller ID & Block` (version 1.56) is a phone-call number identification app which is vulnerable to the public leakage attacks. The unprotected `.provider.CustomSuggestionsProvider` component allows attackers to query if a particular contact name is present in the contact list.

`Kii Keyboard` (version 1.2.22r6) is an alternative Android keyboard app vulnerable to public pollution and leakage attacks. This application has a content provider `.enhanced .BlacklistProvider` which is exported but not protected by any permission. This provider contains the blacklist of words which should not be shown as predictions. The attacker can manipulate or steal the content of the blacklist. `AppLoc` (version 1.99.2) is another app which allows users to lock apps with passwords. This app is vulnerable to the public leakage attacks due to an exported content provider which leaks information about the locks such as the process names and alarm times. The attacker can also apply SQL injection to retrieve information from the tables such as locations by setting the `projection` argument to `"* from locations;"`.

`ES File Explorer File Manager` (version 3.1.2), a popular file manager app, is vulnerable to public file access attacks. The exported content provider, `app.FileContentProvider` allows the attacker to obtain the file descriptors for arbitrary files in the private internal or external storage which can lead to the leakage of sensitive information.

`Lelong.my - Shop and Save` (version 1.2.5.3) is a shopping app, vulnerable to the leakage and pollution private attacks. The `shoppingcart.PaymentOptionsActivity` activity in this app allows the attacker to pop up order requests where the price and order ID can be manipulated by the attacker. Another activity `product.ProductNewActivity` allows the attacker to perform `rawQuery()` and `insert()` to the database to search items. `Money Manager Expense & Budget` (version 2.4.6) is a money manager app vulnerable to private leakage attacks. It exposes an activity allowing attackers to query and pop up bills for a specific account id. `Adidas World Football Live WP` (version 3.1) is a sport news app which is vulnerable to private pollution attacks. It has an exposed broadcast receiver which can be exploited to insert scheduled advertisements into its private database.

Effectiveness of the Symbolic Models. `chomp SMS` (version 6.07) is an SMS app vulnerable to pollution public attacks. This app requires accurate modeling of the `android.content.UriMatcher` and `android.content.ContentUris` libraries. The `provider.ChompProvider` component is vulnerable and accepts requests to update the scheduled messages if the URI parameter of the `update` API passes certain constraints. The goal is to generate specific values for each parameter of the `update` API (e.g., the URI parameter) to use in a working exploit. Our model for the `UriMatcher.match(Uri)` method tries to find a registered URI matching the given URI parameter: (1) our model checks if the URI parameter's authority is `com.p1.chompsms.provider.ChompProvider`; (2) it checks the path segment of the URI. If it is `scheduled_messages`, all the scheduled messages can be updated with the playloads crafted by the malware. In this case, DBDroidScanner generates the corresponding constraints using our model. Solving the constraints gives the attack URI parameter: `"content://com.p1.chompsms.provider.ChompProvider /scheduled_messages"`. Otherwise, if the path segment matches `"scheduled_messages/#"`, `ContentUris.parseId(Uri)` is invoked to retrieve the last path segment of the URI parameter and use it as the selection argument for the `SQLiteDatabase.update` sink method. In summary, the constraints generated using our models constrain the URI parameter to contain the `scheduled_messages` path segment and its last segment to be a number. Solving these constraints, DBDroidScanner generates a malicious URI, `"content://com.p1.chompsms.provider .ChompProvider/scheduled_messages/1"`, which triggers a different execution path.

We now discuss how we generate an accurate exploit for the `com.netease.cloudmusic` (version 1.7.3) app which is vulnerable to private leakage attacks. The vulnerable component to which malware sends an Intent exploit is `LoadingActivity`. The victim app obtains the URI object set in the Intent by invoking the `Intent.getData()` method and performs the following validations: (1) the scheme part of the URI object is checked whether it is `"content"`. Our analysis also generates the corresponding constraints using our model for the `android.net.Uri` library; (2) the `ContentUris.parseId(Uri)` method is called with the URI object passed as the argument to obtain its last path segment and checks if the last path segment is a number. Our model for `android.content.ContentUris` generates the corresponding constraints and our tool generates `"content:///1"`[10] as the malicious URI part of the Intent message. The Intent is further configured by setting its `action` and `type` attributes to target the victim app and launch a private leakage attack.

B.1 Representative Applications in [31]

Public Database Attacks

DBDroidScanner did not find public database vulnerabilities in the updated versions of: `Shady SMS 4.0 PAYG`, `Nimbuzz Messenger`, `MiTalk Messenger`, `Youdao Dictionary`, `Dolphin Browser HD`, `Mobile Security Personal Ed.`, `Youni SMS`, `Netease Weibo`, `Sina Weibo` and `Tencent WBlog`. Our results are consistent with the results reported by ContentScope on the following apps (these apps have not been updated): `360 Kouxin`, `GO FBWidget`, `Boat Browser Mini`, `Droid Call Filter`, `GO TwiWidget`.

For the remaining apps in Table 2, we present the public database attack case-studies as follows. The results are sorted by the application ID in the ID column.

1. `Pansi SMS` is a messaging app which is vulnerable to the leakage and pollution attacks. ContentScope reports that the `.provider.MsgSummaryProvider` is vulnerable. However, this component is protected by both `readPermission` and `writePermission` in the updated version of the app. Therefore, malware will not be able to access this component. However, DBDroidScanner finds other content provider components which are not explicitly protected. The target SDK version for this app is 13 (<17), hence the content providers are exported by default and this application is vulnerable in all compatible Android platforms. The vulnerable content providers are: `.provider.PansiContactProvider`, `.provider .PhraseProvider` and `.provider.SmsDraftProvider` which may be compromised by attackers to steal or manipulate information such as contact details.

3. `mOffice - Outlook sync` is a productivity app which synchronizes private contact information with remote desktops.

[10]The authority part of the URI can be empty.

The updated app studied in our work does not export .dao .DBProvider anymore but .dao.BackupProvider is another content provider vulnerable to leakage and pollution attacks. Thus, attackers can steal and manipulate the sensitive data such as SMS and contact details.

6. The updated version of GO SMS Pro, an instant messaging app, is vulnerable under Android 16 and below. The .StaticDataContentProvider and .golauex.smswidget.SmsProvider components are unprotected content providers vulnerable to the public leakage and pollution attacks.

7. In the Messenger WithYou app, we find that the openFile() method in the MiyowaExplorerContentProvider content provider may return arbitrary database file descriptors, which is similar to ContentScope. In addition, we find the openAssetFile() API as another source method to trigger a similar vulnerability.

14. We could only find an updated version of the Maxthon Android Web Browser app. We find SQL injection and pollution vulnerabilities in the .BrowserProvider content provider if it runs in Android 16 and below. Even though ContentScope reports that the vulnerability in this app is fixed, the exported attribute of the vulnerable provider in the manifest file is not explicitly set to "false" in the updated version, nor is it protected by any permission. Hence, DBDroidScanner still reports vulnerabilities which are consistent with descriptions provided in [31]. These attacks can successfully be launched in Android SDK $\leq$ 16.

18. The Tc Assistant app logs outgoing calls and traffic. The .net.provider.TrafficProvider and com.wali.android .provider.LogsContentProvider content providers in the updated version of this app are not protected resulting in sensitive data leakage.

Private Database Attacks. Now we present the private database attack case-studies for the vulnerable apps in Table 2. We remark that ContentScope only aims to detect the public database vulnerabilities, however, our analysis also finds the private database vulnerabilities in these representative apps analyzed by ContentScope. The last column in Table 2 shows our results for the private database vulnerabilities. The dynamic testing for a few apps could not be done due to some practical problems: e.g., we have to register a valid phone number for one app or the app crashes once launched due to incompatibilities. Thus, we do not confirm if these apps are vulnerable or not. We now discuss the case-studies by the ID column of Table 2.

1. Pansi SMS is vulnerable to private pollution and leakage attacks. The attacker can launch privilege escalation attacks using the vulnerable MrBeanUpgradeMsgActivity activity in this app to update the SMS content. It is also possible to exploit a vulnerability in SearchActivity activity to force this app to search the messages in the phone for a keyword.

3. The mOffice - Outlook sync app which has SMS read and write permissions can be reached by a malware through its com.innov8tion.mobisynapse.core.SMSReceiver component. By sending a malicious Intent, the malware can choose an SMS ID and launch a privilege escalation attack by forcing the app to send an update request to the content provider of the default SMS app on the phone. Malware can also force the mOffice - Outlook sync app to send query requests to .dao.DBProvide even though this component is not exported and has been protected by readPermission. The malware can craft malicious Intents and choose arbitrary task identifiers (e.g., taskID) to invoke the vulnerable component .activity.task.EventEditActivity to launch leakage attacks.

4. A malware on the phone can launch privilege escalation attacks and force the Shady SMS 4.0 PAYG app to manipulate a particular SMS content.

7. Messenger WithYou is vulnerable to the private pollution and leakage database attacks. For instance, the malware can send malicious Intents to the CapptainWebAnnouncementActivity component to send query or delete requests using arbitrary WHERE clauses.

11. The GO FBWidget app has a vulnerability which can be exploited to send requests to manipulate Facebook authentication data.

12. An existing malware on the phone can force the Netease Weibo app to open an input stream for a given URI.

13. Although Dolphin Browser HD is updated to fix its public database vulnerabilities, it happens to be vulnerable to private leakage attacks. The vulnerable component is the AddBookmarkPage activity which allows malware to force this component to send query requests to the BrowserProvider even though the latter is not exported in the updated version of the app. Therefore, the attacker can access the component which was reported by ContentScope [31] to be vulnerable in a new way.

19. GO TwiWidget is another app which can be forced by the malware to query the details of an account on the device.