Don't Touch that Column: Portable, Fine-Grained Access Control for Android's Native Content Providers

Aisha Ali-Gombe, Golden G. Richard III, Irfan Ahmed and Vassil Roussev
Department of Computer Science, University of New Orleans
New Orleans, Louisiana, USA
aaligomb@uno.edu, golden@cs.uno.edu, irfan@cs.uno.edu, vassil@cs.uno.edu

Abstract

Android applications access native SQLite databases through their Universal Resource Identifiers (URIs), exposed by the Content provider library. By design, the SQLite engine used in the Android system does not enforce access restrictions on database content nor does it log database accesses. Instead, Android enforces read and write permissions on the native providers through which databases are accessed via the mandatory applications permissions system. This system is very coarse grained, however, and can allow applications far greater access to sensitive data than a user might intend.

In this paper, we present a novel technique called *priVy* that merges static bytecode weaving and database query rewriting to achieve low-level access control for Android native providers at the application level. *priVy* defines access control for both database schema and entities and does not require any modifications to the underlying operating system and/or framework code. Instead, it provides a new Controller stub which is statically woven into the target application and a Controller interface for setting access levels, thus making it accessible and easily adoptable by average users. We provide an evaluation in terms of the resilience of applications to instrumentation as well as static and runtime instrumentation overhead. In our testing, *priVy* incurs an average of 1032 additional method calls or joinpoints created and it takes an average of 15 seconds to recompile an app and imposes virtually no runtime overhead.

1. INTRODUCTION

Smartphone technology has not only revolutionized our telephony experience, but has successfully integrated a vast amount of personal data, including our address books, calendars, diaries, pictures, etc. onto a single device. From a security perspective, the ease and convenience provided by this integration can have disastrous consequences, serving as a single point of exposure for a tremendous amount of personal data if not properly managed.

WiSec'16 , July 18-22, 2016, Darmstadt, Germany

DOI: http://dx.doi.org/10.1145/2939918.2939927

Access to data and resources on Android systems is regulated by two important concepts, specifically, the permissions model and application sandboxing. Third party applications are required to make explicit requests for permissions at installation time and while this mechanism provides a general idea of what an application can access on a device, it does not provide the ability to institute fine-grained control over sensitive data. Essentially, it's an all-or-nothing model under which the user has to approve all permissions or abort the installation of the application. Perhaps more disturbing is that the approved permission(s) remain a right of the installed application as long as it remains on the phone.

Android extends this model to cover structured data stored in SQLite databases. However, it does not separate roles and privileges on the database, nor does it protect content data at the schema or entity levels. In fact, it does very little to protect the privacy of the stored user data and its associated metadata. Such a wide level of access is tantamount to giving the application administrative rights over the target provider. For example, this system does not distinguish accesses to a contact's phone number from the email and physical addresses. Other important information like the "last time contacted" as well as account type and names are also easily accessible with a simple READ permission. Similarly, write permission on the contacts provider allows an application to insert, delete and modify any contact at will. The application can also create groups and make them invisible. Such "perceived" benign access however can lead to malicious contacts been created and synched to restricted groups in major accounts like Google.

Looking at the bigger picture, the privacy violation scales beyond the device user alone. It also exposes data associated with third parties to the prying eyes of malware and other privacy-violating applications. For instance, clearly mapped information like phone numbers, email and physical addresses provide sufficient information about third parties that they could be used to support targeted advertisement, social engineering, surveillance, and physical attacks

Furthermore, due to the interconnectivity of the different providers and their data, we have found the current approach to result in various forms of inferential permission leaks. Other security breaches like denial of service due to malformed SQL data are also possible.

To reduce the propensity of these problems and provide users with additional control over the Android content providers, Mutti et al [16] proposed an integration of SQLite and SELinux. Based on context security, this system enforces fine grain access control at the lowest level in the database. Unfor-

tunately, the solution requires extensive changes in the operating system code to accommodate the security context schema table and its corresponding library code. Given how long it takes for Android to effect changes and for manufacturers to integrate such solutions on existing systems, techniques that require extensive OS modifications are not viable, practical options for an average user.

To solve these problems, we developed a new technique that enforces access restrictions, query-rewriting and database access logging via static bytecode weaving. Our system, called *priVy*, does not require any change to the Android kernel and middleware. Furthermore, *priVy* does not treat SQLite databases as a single information store with a single set of access permissions; instead, it enforces restrictions for different schema and entity levels by re-implementing content provider library code using instrumentation at the application level, based on user-provided access restrictions. The new weaved checking code forces the application to access only user-approved schema or entities, while maintaining application integrity.

Contributions Our techniques provide the following unique features:

- *More control over data*: *priVy* provides schema and entity level access control for Android content providers through method hooking, access constraint enforcement, and query re-writing.
- *Portability*: Our scheme does not require changes to the Android kernel and/or framework code, for maximum usability.
- *Efficiency and Usability*: Aside from minimal instrumentation overhead and better control over user data, the use of *priVy* is virtually transparent.

The rest of the paper is organized as follows: Section 2 presents background on SQLite and Android content providers; Section 3 provides an overview of possible threats and vulnerabilities on the content provider; Section 4 details the design of *priVy*; Section 5 and 6 presents the implementation and evaluation of our work respectively; Section 7 reviews the related literature followed by section 8 that concludes the paper.

2. BACKGROUND

2.1 SQLite Databases

SQLite is a single-user relational database management system (RDMS) used for storing structured data. Unlike a traditional RDMS, SQLite is a server-less database engine that stores data in normal files. It manages access and concurrency based on direct file reads and writes and operating system-level file locks, respectively. SQLite is lightweight and efficient and requires little configuration, making it the database engine of choice on many operating systems, such as Android and Apple's iOS.

2.2 Android Native Providers

Android offers built-in native content providers that store a variety of user data maintained by the system. Each is associated with at least one SQLite database that contains various tables, columns and entities. Some of the Android native providers are: Contacts, CallLog, VoiceMail, Browser, Settings, Media and Dictionary.

These providers together with the content resolver provide the basis for Android CRUD (Create, Read, Update, Delete) operations, corresponding to SQL insert, query, update and delete operations on database objects. The chain of events for data access occurs at two levels. At a high level, access begins with the resolver object invocation of one of the CRUD functions, passing at least a *Uri* parameter, which identifies the location of the required data. Other parameters for CRUD functions include column name(s), a WHERE clause, and order information. The resolver validates the *Uri* and then passes the request to its provider. The provider performs permission checks and if the requesting application has the required permissions, it uses the function parameters to construct an appropriate *SQLiteStatement*.

At a lower level, the SQLiteStatement is passed to the native content provider through the binder parcel. The native library translates the parcel and sends the request to the database engine, which then performs syntax and semantic checks, expansion and code generation. The result is sent back through the same route. In the case of read operations, a database *Cursor* is returned, For write operations, an integer indicating the number of entries affected is returned.

As discussed above, each of the CRUD operations triggers permission checks by the content provider. Queries are protected by READ permission while insert, delete, and update are guarded by WRITE permission. However, these coarse-grained permissions do not distinguish database roles for applications or privileges for individual data items. A simple READ permission allows access to all the tables, column and rows in the entire database, while a simple WRITE permission allows manipulation or deletion of any database entities.

3. THREATS AND VULNERABILITIES

The coarse-grained access control for databases under Android has serious security and privacy implications. In our preliminary research, we analyzed the contacts database and explored some issues associated with providing arbitrary applications with READ and WRITE access to this database.

3.1 Security Implications

Denial of Service: We explored a vulnerability with account types based on a malformed SQL statement that can crash the *acore* process, resulting in denial of service on the phone. A malicious app with WRITE permission can create a new contact without the user's knowledge under the "com.google" account type with a malformed account-name which contains a SQL terminator ";". The system will accept the malformed account name at the time of insertion, but after a while, Android will try to synchronize and delete bad account names. When this occurs, the malformed account-name will trigger a SQL exception in the SELECT statement and crash both the contacts application and the *acore* process. This key process is designed to automatically restart after it is killed, however, the malformed name will cause it to die once again. The repeated restart followed by crash of *acore* results in a denial of service attack on the phone and the only solution is to delete the entire contacts database, causing a loss of all local contacts if the user has no backups at hand.

Permission Leak: We also discovered that applications with the READ-CONTACTS permission can infer the user accounts on a device without requesting the GET-ACCOUNT

permission. If a contact belongs to an account, the account name will be written alongside the contact in the RAW-CONTACTS table. And since there is no restriction on schema or column, an application can read the account name and type for all the contacts on the phone.

Malicious Contact: Finally, an application with WRITE access to the contacts can add a new contact under a particular account and group without restriction. When that account is synced, the contact gets pushed on to the server. This becomes a serious problem if, for example, the contact is pushed into an important work group that shares confidential information or if a contact's email address is secretly updated, to facilitate a targeted attack.

3.2 Privacy Concerns

Applications with appropriate coarse-grained permissions can read clearly mapped data containing names, phone numbers, email and physical addresses, and even IM status. This data can clearly distinguish an individual and be used for annoying advertisement or targeted social engineering attacks. Worse, information such as "last time contacted" can provide inferential information about call logs without having the CALL-LOG permission.

3.3 Forensic Concerns

With WRITE permission, update, insert, and delete database operations can be performed by an application with very little data available to support attribution, since Android produces no audit logs associated with database operations. This is primarily because SQLite is a single user system and is not designed to keep track of who performs what operations on a system. For forensics investigation, this makes it very hard to ascertain if a particular entry in the database is added or updated by the user or by a malicious application.

4. SYSTEM DESIGN

Our goal is to define low-level access controls for Android's native content providers and enforce these access controls for third party applications. This will ensure that users have tight control over read/write accesses on sensitive data for instrumented applications.

priVy is comprised of two components, a Controller app and Controller stub. The Controller app is an independent application running in a different process that registers an instrumented application and sets up and manages its access levels. The stub provides the weaved code that forces the instrumented app to verify access levels at startup, enforce access constraints, perform query-rewriting as necessary, and effect database auditing. The architecture of *priVy* is illustrated in Figure 1.

4.1 Controller Stub

Our approach uses the AspectJ instrumentation framework [14] to insert and enforce fine level access verification, query re-writing and database auditing for Android's native providers. AspectJ is an aspect-oriented programming (AOP) extension developed specifically for Java that allows cross-cutting concerns defined as aspects to be statically weaved into either raw Java source code or a compiled class. Our system employs a well known AspectJ compiler called *ajc* [1] to perform the application repackaging of Android binaries with our specially crafted, modularized aspects. Unlike most instrumentation libraries, AspectJ can perform highly optimized code injection at runtime, making it an ideal choice for our system. It can manipulate parameters, return values, and target objects, and new code can be inserted to run alongside or replace an existing method implementation based on some static or runtime condition.

In aspect-oriented programming, a *joinpoint* is an identifiable construct within program execution, e.g., a method call, while a *pointcut* is a program element which defines a joinpoint via a signature pattern. These signatures can contain *modifier, type, id* or *throw* patterns. The weaving process in static instrumentation creates and manages joinpoints based on these predefined signatures at compilation time. When a certain joinpoint is reached at runtime, AspectJ executes the encapsulated analysis routine called the *advice* defined for it. In *priVy*, this contains the newly inserted policy and query-rewriting code.

Depending on the cross cutting concern, signatures can be made very broad using wildcards or specific with direct package names, return types and parameter types. In *priVy*, we designed signatures for Android packages related to data access on SQLite databases. The three most important are *Database, Content Provider and Resolver*. The database package hosts the main SQLite database object and corresponding methods to query it in raw form. It also provides the *Cursor* interface for reading the results of database queries. It is important to note that Android does not support direct raw access for databases associated with an application with a different *uid*. Access to such data can only be provided via the *Uri* of the target Content provider. The provider classes expose data of one app to the code executing in a different process.

Generic AspectJ advices were then developed around the methods in the relevant classes from the packages discussed above. These advices are encapsulated in an aspect which is then statically recompiled into an Android binary. The result is the same Android binary extended with our controller stub. This static instrumentation process intercepts the resolver CRUD functions and inserts the controller code where necessary. As mentioned in Section 2, direct access to native databases is completely prohibited by Android and access is only available through the exposed native content. Thus, it is relatively convenient for us to develop specific signatures corresponding to only the resolver and provider packages.

As shown in Table 1, insertion operations can be performed in three different ways, either via a single insert, bulk insert, or using a content provider operation. Delete, update, and query operations can each be performed in two different ways. Our signatures take into account all these and we target the respective joinpoints accordingly.

AOP exposes three different ways to weave the checking code, either as "before", "after" or "around" advice. The "before" advice inserts new code before the joinpoint, thus its execution precedes that of the joinpoint. The "after" advice prepends the code beneath the joinpoint's code, while the "around" advice inserts it within the joinpoint's implementation. With the exception of adding auditing log entries, where code is inserted using "after" advice, all other code that performs constraint checks and query-rewriting uses "around" advice, which can perform code injection in the middle of method execution and allows manipulation of target object, parameter(s) and return value. This enables us to generate the correct return values in case a query is

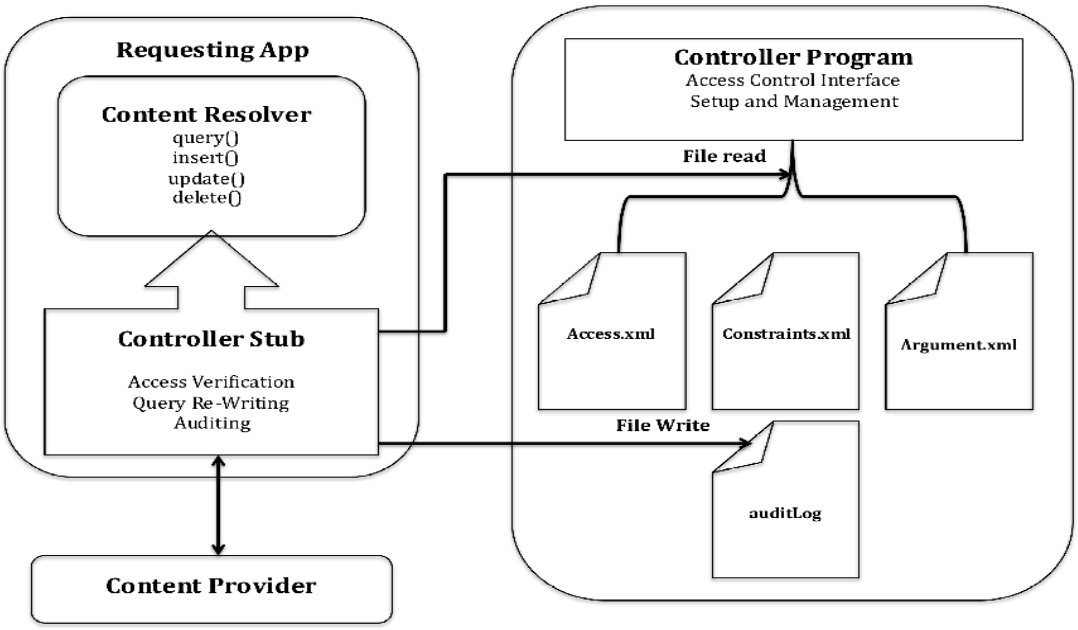

Figure 1: *priVy's* System Architecture.

blocked or restricted. It also allows us to enforce constraints and reflectively perform new method invocation on an already created object residing in memory.

4.1.1 Access Verification

At runtime, when the instrumented app begins execution, the controller stub performs the access verification as illustrated in Figure 1. It reads and parses the assigned access control for the target application from the world readable, shared preferences XML file for the controller app. It sets up the global variables for the access level, schema, and column as well as entity privileges for each provider. The global variables are used by the CRUD operation's joinpoint to determine how the method call will proceed when invoked.

The CRUD function's access level can be ALL_ALLOW, ALL_BLOCK and RESTRICT. The ALL_ALLOW access level, as the name implies, does not impose sanctions on the joinpoint and simply allows it to proceed with its original parameters. ALL_BLOCK, on the other hand, completely blocks the execution of a joinpoint. For ALL_BLOCK, our controller stub must ensure that the query is actually blocked while not affecting application stability. We are able to achieve this by ensuring the affected functions return appropriate values as shown in Figure 2 for Query joinpoint. Specifically, depending on the type of access functions, different actions must be taken:

1. Query and Insert: We transform the given *Uri* parameter of the function into an *Entity Uri* with appended zero. For queries, the system proceeds with this special entity *Uri* which in turn will force the return of an empty cursor with at least header information. For insert operations, the entity *Uri* gets returned and the parameters are discarded.

2. Update and Delete: Functions performing update and delete operations expect an integral return type indicating how many rows were affected. We simply return 0, indicating that no rows were affected and thus the program will continue to execute smoothly.

The RESTRICT option regulates access to database schema, column and entities. In a relational database, a schema represents a logical group of objects. In this paper we restrict the schema definition to the database tables available through the content *Uri*, e.g., the contact table in "Contacts.db" or the events table in "Calendar.db". Also, we logically include all objects in a table that can be grouped by the same MIME types, like emails, phone numbers, and addresses, as different schemas.

Thus, the schema restriction ensures an app only queries from the approved tables or MIME type(s). Since most of these MIME types have individual Uris assigned to them through the *CommonKind Uri*, their schema restriction must ensure a entity restriction on the main table as well. The controller stub makes a decision to ALLOW, BLOCK or REWRITE the query based on the schema restriction established by the user. For example, if a user has a schema restriction set up to only allow access to email information and and the app requests both email and phone numbers, *priVy* must re-write the query such that only the email table gets projected on the SQL statement. Furthermore, restrictions can be imposed on database columns so that certain columns are prohibited from being viewed by apps, e.g., account-type and name, or an entity based on a column value.

Aside from the two mandates mentioned above for BLOCK option, a third condition becomes necessary here, specifically, that *priVy* must ensure that any other part of the application that depends on the return value of the function does not crash. This is mostly an issue with query functions, because they return a cursor and the program may have been designed to access a particular column which may not

Table 1: Joinpoints Picked by *priVy*'s Pointcut Signatures

Target Object	Insert	Update	Query	Delete
Content Resolver	insert(..)	update(..)	query(..)	delete(..)
Content Resolver	bulkInsert(..)			
ContentProviderOperation	newInsert(..)	newUpdate(..)	newAssertQuery(..)	newDelete(..)

```
pointcut getCurObj(Uri uri, String[] Projection,
      String Selection, String[] Selection_Args):
      call(*..*Cursor* *..*.query(..))
      || call(*..*Cursor* *..*.*Query(..)))
      && args(uri, Projection, Selection,
      Selection_Args,..) && NotNewLogger();

Object around(Object tar, Uri uri, String[] Projection,
      String Selection, String[] Selection_Args):
      target(tar) && getCurObj(uri, Projection,
      Selection, Selection_Args){
      //...
      //...
      ContentValues cont = getAccess();//From SharedPrefs
      if (cont.containsKey(uri.getAuthority())){
            start = System.nanoTime();
            String level =
            cont.get(uri.getAuthority()).toString();
            if (level.equals("ALL_ALLOW")){
                  ret = proceed(tar, uri, Projection,
                  Selection, Selection_Args);
            }else if (level.equals("ALL_BLOCK")){
                  ret =  proceed(tar, getEntityUri(uri),
                  Projection, Selection, Selection_Args);
            }else if (level.equals("RESTRICT")) {
                  checkSRestrict(..);//Schema Restriction
                        //...
                  checkCRestrict(..);//Column Restriction
                        //...
                  checkERestrict(..);//Entity Restriction

            }else{

            }
      }
```

Figure 2: Advice on a Query Joinpoint that Shows How the Controller Stub Performs Access Verification

be available due to restrictions. To solve this problem, we instrument all the functions that access cursor information directly. The advice on these joinpoints tests if the column requested is available and if it isn't, the column will return a empty string. This has proven to work well in practice to ensure that applications do not crash due to the imposed access restrictions.

4.1.2 Query Re-Writing

Android creates a proper SQLstatement after the request has passed the permission checks. Since our system operates at the highest level, we rewrite the intended query by altering and/or supplying new CRUD function argument(s). These functions contains *Uri*, *Projection*, *Selection Selection-Arguments* and *Content Values* parameters.

In SCHEMA restriction, *priVy* compares the query Uri with the restricted schema, if matched, the query is simple blocked otherwise the system allows it to execute. The code snippet is shown in Figure 3.

The query-rewriting module is triggered when the initial query is projected on column(s) and/or entities outside its access restriction. In a query function a projection argument can take an array of column names or null (indicating all rows in a table should be returned). Armed with the

```
// qSRestrict contains list of restricted Uri
public Uri checkSRestrict(ArrayList<String> qSRestrict,
      Uri uri, ContentResolver resolver){
      if (qSRestrict.contains(resolver.getType(uri))){
            uri= getEntityUri(uri);
      }else{
            //...
      }
            //...
}
```

Figure 3: Schema Restriction Check on a Query Function

column-level access restriction, the Controller stub executes *checkCRestrict* function and re-writes the query based on the following rules;

1. **If projection is not null** - the stub checks for the intersection of the *projected column(s)* and the restricted column(s) and then removes them from the projection list as shown in Figure 4.

2. **If the prohibited column is the only column to be projected** - the function will be blocked completely. This is because exchanging the prohibited list with null will return all the columns including the prohibited ones.

3. **If projection is null** - For query, the stub checks the intersection of the columns of the *return cursor* and the restricted column(s). If found, the intersected column(s) are removed and the query continues with the remaining column as shown in Figure 5. For Update and Insert, restricted column are prevented from database write thus key sets of the content values are compared against the restricted column and removed if there is an intersection. Delete operations do not require column projection.

```
if (Projection!=null){
      if(myMap.keySet().contains(resolver.getType(uri))){
            String val = myMap.get(resolver.getType(uri));
            for(String str: Projection){
                  if(!(resolver.getType(uri)+str).equals
                        (resolver.getType(uri)+val)){
                        newProj.add(str);
                  }
            }finProj = newProj.toArray(new
            String[newProj.size()]);
      }else{
            finProj= Projection;
      }
}
```

Figure 4: Column-level Restriction with Not-Null Projection

Selection and Selection-Arguments indicate the WHERE clause column(s) and value(s). The user can restrict access on some predefined values, e.g., to certain account types, whitelisted contacts, etc. For the most part, we don't test or nullify these arguments, but rather we enforce the new

```
if (ret instanceof Cursor){
     Cursor cur =(Cursor)ret;
     if(cur.getCount()>0){
          String[] pNames = cur.getColumnNames();
          if(myMap.keySet().contains
               (resolver.getType(uri))){
               String val =
                    myMap.get(resolver.getType(uri));
               for(String str: pNames){
                    if(!(resolver.getType(uri)
                         +str).equals(resolver.getType
                         (uri)+val)){
                         newProj.add(str);
                    }
               }finProj = newProj.toArray(new
                    String[newProj.size()]);

          }else{
               finProj= null;
          }
     }
}
```

Figure 5: Column-level Restriction with Null Projection

specification by concatenating our restriction to an already established WHERE clause. For instance, an application might be restricted to only query contacts from account-type *"com.google"* and we simply ensure that this is enforced by influencing the WHERE clause. If this restriction involves only one entity, the controller stub appends it with an "AND" operator to the function's WHERE clause, if not null. If the WHERE clause is null, however, the stub then substitutes the null with the new restriction, and the function proceeds with this new value. On the other hand, the situation is more challenging when there are more than one entity restriction and it applies to different tables (e.g., account type (Raw_Contacts) and lookup key (Contacts)). In typical SQL we can perform complex joins on the different tables. However, on content providers such operations are very limited. To solve this, we extract the primary key (and foreign key where necessary) from each of the tables and use them as the parameter(s) for the target query's WHERE clause as shown in 6.

```
public String checkERestrict(Uri uri, ContentResolver resolver){
     String finSel= null;
     if (uri.getAuthority().contains("contacts")){
          // for each entity restriction,
          // getContactIds(..) gets its primary key column.
          //The intersection of the results is return in fin
          String fin = getContactIds(resolver);
          if(fin!=null){
               if(uri.equals(ContactsContract.Contacts.
                    CONTENT_URI)){
               finSel = ContactsContract.Contacts._ID +
               " IN ( "+fin+" )";
               }else if (uri.equals(ContactsContract.Data.
                    CONTENT_URI)){
                    finSel = ContactsContract.Data.
                         RAW_CONTACT_ID + " IN ( "+fin+" )";
               }else{
                    finSel = ContactsContract.
                    RawContactsEntity.CONTACT_ID + " IN (
               "+fin+" )";
               }
          }
     }
     return finSel;
}
```

Figure 6: Code Snippet Showing Entity Restriction for Contacts Provider

For example, consider the query "_ID from contacts WHERE lookup key = value" and the query "_ID and RAW_CONTACT_ID from raw_contacts WHERE account_type = value". The intersection of _ID and RAW_CONTACT_ID in these query results will be the new WHERE clause for the target CRUD operation.

For delete and update functions, a developer may or may not supply the WHERE clause and/or its argument. According to a user's preferences, our system can enforce restrictions on when and where delete operations can occur by reflectively invoking a new delete function within the joinpoint on the target object. After it returns, the new return value is supplied as the return value of the joinpoint's advice as shown in Figure 7.

```
pointcut deleteInst(Uri uri):call(* *..*.delete(..))
     && NotNewLogger() && args(uri,..);

Object around(Uri uri, ContentResolver tar):
     deleteInst(uri) && target(tar){
     if (access.containsKey(uri.getAuthority())){
            //
          }else if (level.equals("RESTRICT")){
               ContentResolver resolver = null;
               if (tar instanceof ContentResolver){
                    resolver = (ContentResolver)tar;
               }
               if (resolver!=null){
                    //check Schema Restriction
                    uri = checkSRestrict(qSRestrict,
                         uri, resolver);
               }
               if (!uri.toString().contains("/0")){
                    Log.d(uri.toString(), "here3");
                    //check Entity Restriction
                    String finSel = checkERestrict(uri,
                      resolver);
                    if(finSel!=null){
                         //populate selection
                    }
               String[] selArgs = (String[])args[2];
               Object[] params = new Object[]{uri,
                    sel, selArgs};
               Class clazz = thisJoinPoint.getSignature().
                    getDeclaringType();
               //Reflectively Recreate Delete
               //function with new selection and
               //return number of rows deleted
               try {
                    String methName = thisJoinPoint.
                    getSignature().getName();
                    Class[] paraTypes
                         =getMeth(thisJoinPoint);
                    Method method
                         =clazz.getDeclaredMethod(methName,
                         paraTypes);
                    ret = method.invoke(tar, params);
                    delRet= true;
               }catch (Exception e){
                  //
               }
          //
     }
     return ret;
}
```

Figure 7: Code Snippet Showing Query Re-writing for Delete Function

4.1.3 *Database Auditing*

Currently, Android does not provide any form of auditing on the native provider. As discussed in Section 2, applications are considered individual users with different user IDs. It is important to keep track of which applications perform which actions on system resources, especially since these applications are typically created by different developers and may manipulate the same data with few restrictions.

In our prototype implementation of *priVy*, we introduce auditing using a file attached to the Controller app called

```
after(Uri uri, String[] Projection, String Selection,
      String[] Selection_Args) returning (Cursor ret):
      getCurObj(uri, Projection, Selection, Selection_Args){
      //...
      if (ret.getCount()>0 ){
            String vals=null;
            StringBuilder stb = new StringBuilder();
            if(Projection!=null){
                  for(String str: Projection){
                        stb.append(str);
                        stb.append(",");
                  }

            }
            vals = stb.toString();
            String args=null;
            stb = new StringBuilder();
            if(Selection_Args!=null){
                  for(String str: Selection_Args){
                        stb.append(str);
                        stb.append(",");
                  }
            }
            args = stb.toString();
            String audit = "Time"+Long.toString(System.nanoTime())
            +" Uri "+uri.toString() +" Values "+vals + "Selection"
            +Selection + " Selection_Args "+ args+
            thisJoinPoint.getSignature().getDeclaringTypeName()+
            "."+thisJoinPoint.getSignature().getName();
            Log.d("R-DAC","Query Audit- "+audit);
            //…
}
```

Figure 8: Instrumentation Code Snippet for Auditing Query Operations

the auditLog. We implement this by injecting the auditing function after the CRUD function has executed and returned a desired result. For insert, an "after" advice will request for the returned *Uri* and then parse it get the row id. This package name, row id, together with Uri name, Content Values and time stamp are written to the auditLog file.

On update, the return value is the number of rows affected rather than the *Uri*. Thus, we need a global variable to keep track of the row lookup ids (rowid) affected by the update function. We use this global rowid, together with package name, Uri name, Selection and its Arguments (if any), Content Values and time stamp as an audit file entry. This also applies for Delete operations. Query operations return a Cursor, thus we keep the audit of the query parameters as well as the number of rows in the cursor. We do not track the IDs of the columns because it may or may not be part of the projection list. The code snippet for auditing query operations is shown in Figure 8.

Apart from its major objectives, our controller stub further checks for malformed strings in arguments passed to the CRUD function. This is important so as to prevent the *denial of service* attack mentioned in Section 3. This functionality checks for special characters in the content value(s) of an insert or update function. It then triggers warning to the user and he/she can opt to remove such special character.

4.2 Controller App

The controller app running on a separate process coordinates the content provider restrictions for targeted applications. Our aspects are written as generically as possible to integrate into any Android app as well as work for all the native providers. The controller app provides an interface for choosing the access levels and further access restrictions on schema, column or entity, thus saving the cost of re-instrumentation in case changes need to be made. This significantly improves the usability of our approach.

When an application is installed, the user needs to register it with the controller. Its user interface (UI) exposes the available access level/restrictions for the user to choose from. After selection, the values is set for the target application in a shared preferences XML file maintained by this controller app. The Controller stub queries these files at runtime. The controller app maintains three different XML files for the access level, constraints, and the arguments, as shown in Figure 1.

1. Access.xml - this file contains entries for all registered instrumented apps. It takes the concatenation of package name, provider names and CRUD function name as the key which is also the record identifier *RID*, while the value contains the access level as ALL_ALLOW, ALL_BLOCK or RESTRICT. Listing 1 shows an example of key:value pair in Access.xml file.

2. Constraint.xml - If the access level is set to RESTRICT, the schema, column or entity constraint has to be provided. This constraint is registered in constraint.xml. Its entries are the RID as provided in Access.xml file and the values are the constraints separated by commas as shown in Listing 2. Empty brackets indicate there is no constraint on the element. The SCHEMA constraint has to take complete Uri string names, while the COLUMN and ENTITY constraints contains the Uri and column name, each of which can have zero or more constraints.

3. Argument.xml - As mentioned above, the ENTITY constraints are enforced in the WHERE clause of the SQLStatement. Thus for each entry in the Constraint.xml file that contains a record for an ENTITY constraint, there must exist an record in the Argument.xml file that provides the argument value(s). For example, if Constraint.xml contains a record as shown in Listing 2, the Argument.xml file will have a corresponding record as shown in Listing 3. This constraint ensures an app is restricted from querying contacts WHERE account_type is *"com.google"*.

Listing 1: Entry in Access.xml

```
key - com.bbm:contacts:query:
value <RESTRICT>
```

Listing 2: Entry in Constraint.xml

```
key - com.bbm:contacts:query:
value < SCHEMA(),
      COLUMN(vnd.android.cursor.dir/contact:
        display_name),
      ENTITY(vnd.android.cursor.dir/raw_contacts:
        account_type)
      >
```

Listing 3: Entry in Arugument.xml

```
key - com.bbm:contacts.query:
value < ENTITY((vnd.android.cursor.dir/raw_contacts:
        account_type) :com.google)>
```

Apart from creating and managing access verification information, the controller app also manages the auditLog file.

5. IMPLEMENTATION

We implemented the prototype of our approach in Python, Java and AspectJ as the weaving framework. The Controller app is written as a standalone Android application with three shared preference files that store the access level, constraints, and its arguments for an instrumented app. This app does not require any permissions to install. For the Controller stub, the instrumentation process is implemented using Python scripts which automate application unpacking, repacking and signing, while the weaving aspect is written using Java/AspectJ.

Android apps are shipped as a single zip file called an *apk*, which contains the main *classes.dex* file and other resource files. The classes.dex is a highly optimized compressed file that contains the Dalvik bytecode which is parsed and interpreted by the Dalvik Virtual machine at runtime. It is created by removing redundant information from the app's compiled Java classes. The AspectJ framework, on the other hand, does not understand the Dex file format. Thus, to weave-in the Controller stub, we need to unpack from Dex to Java class files.

The automated instrumentation processing makes use of an open source Dalvik translator called *"dex2jar"* [3] for the unpacking, repackaging, and app signing. This processing is set up on a Linux system with Java and "ajc" compilers installed and the AspectJ library and the Android SDK on its class-path. The weaving module takes the unpacked classes as input which after recompilation executes the repackaging and resigning modules, respectively.

We developed generic aspects that can be woven into any Android application to enforce access control on any of the Seven native providers. However we limited our testing and evaluation to contacts and calendar providers. These two providers contain valuable and sensitive data for both the device user and any third party associated with the user. According to [9], the contact information is by far the biggest privacy concern of all the sensitive data found on smartphones. The contacts provider exposes different kind of data via its numerous Uris. These data are contained in three tables (Contacts, Raw_Contacts, Data) under the contacts database, while the calendar provider on the other exposes five tables (Calendars, Events, Attendees, Reminders, Instance) from the calendar database through its URIs.

Our aspect has a total of 11 pointcuts which corresponds to the 9 method calls as shown in Table 1, one pointcut for application context, and one for the aspect itself. It also has a total of 16 advices for these pointcuts and numerous Java-related methods that help the functionality of the advices.

6. EVALUATION

The target of our evaluation covers two main objectives; Overhead and Application Crash. *priVy* is developed as a User-centric solution with the aim of providing a reliable means of securing and restricting access to native database objects. The goal is to ensure *priVy* works on a diverse group of applications with minimal overhead. More so, we want to ensure the instrumented app does not crash as a result of the weaved controller stub.

We downloaded the top 350 applications from Google Play [2] and choose 76 apps with read/write permission to either the contacts or calendar providers. Our samples are instrumented and repackaged with new sign-in keys. We assess their static overhead in terms of weaving time and number of joinpoints created.

We also measure the runtime overhead, which is the time it takes to execute each of our joinpoint's advice. We developed a test application that triggers all the advices in our aspect (since most of the sample apps trigger only one or two of them) and measured their execution time. Finally we evaluate app crashes by executing each of the instrumented applications. Our testbed is a Samsung tablet running on the Android 4.4.2 kernel. It has some saved contacts under the device's main gmail account, local phone numbers, and others imported from one extra exchange account.

6.1 App Execution

We then test run all the 76 instrumented apps on a three round testing (total 228 execution) on the test bed and examined them for app crashes that can be directly linked to the instrumentation stub. The testing involves changing the different access levels - ALL_ALLOW, ALL_BLOCK and RESTRICT. Each app is manually installed, executed and profile created for those requiring one. We interact with them using touch events, text inputs, and various system events like calls. The testing period for each app ranges between 15 to 20 minutes, depending on the initial setup required by the app.

In the first round of testing, we set the access policies for all the 76 apps to ALL_ALLOW. Our first observation is five apps (Chase, SendHub, All State, BlueBird, Citizens) fail to execute or connect to their server in the first round of testing. An examination into these groups revealed that mostly they are vendor apps like mobile banking. Such apps, for security reasons, do not execute when the signature changes or have broken resources. They fail to execute not because of our instrumentation stub but because of a change in the application file, thus we eliminate these from further testing.

The second group of seven apps (Sirma Bible, Docusign, Autodesk, FaithComesbyHearing, Zillow, Backgrounds, Jiffy) did not make any attempt to query the contacts or calendar database, even though they requested READ and/or WRITE permission. Thus, they were eliminated too.

The final group executed correctly in all the 3 rounds of testing. We randomly change different combination of access restrictions that will trigger the query re-writing module during the manual execution and check for app failure. Within the execution period, we observed that all the 64 apps in this group invoked one or more of our joinpoints. For instance, the BBM app requested READ CONTACT permission and it also asks users explicitly for access to contacts on the setup window. When we BLOCK all contacts, it was not able to access any. Similarly for RESTRICT, it was only able to view contacts from the gmail account. This is the same for other apps like KIK, Pinterest, Mr. Number, AVG AntiVirus, AutoCard, Vine etc. The Sunrise app uses the calendar provider to manage and organize events. We successfully limited the events that can be viewed by this app based on Event_ID.

We have observed that $\approx$ 82% of the apps in our sample perform only database read (query) even though 60% of them request both READ and WRITE permissions.

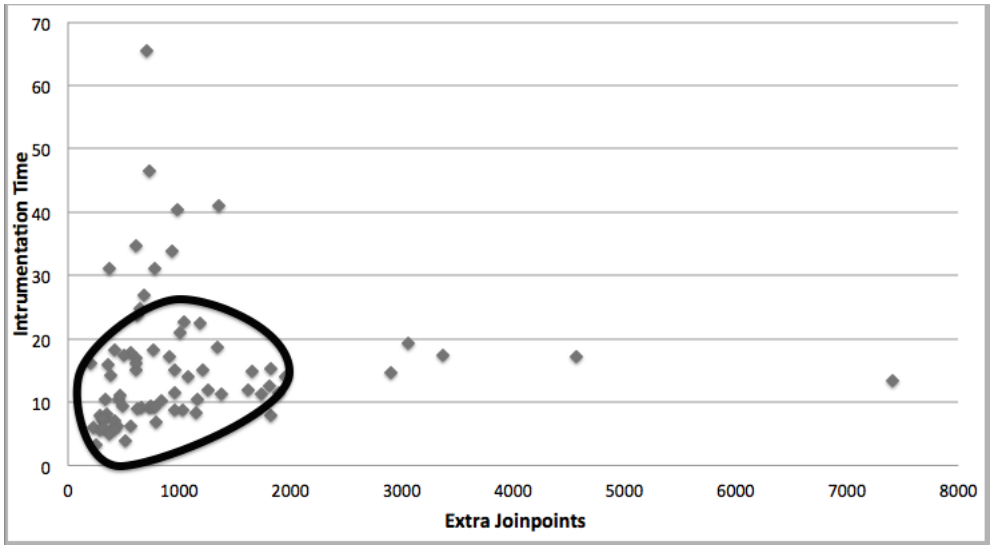

Figure 9: Relationship Between Instrumentation Time and Extra Joinpoints

6.2 Static Overhead

Instrumentation entails weaving new code into a binary and optimization becomes essential in order to avoid bloating the existing code. Specifically, the nature of pointcut signatures can have adverse effects on the number of joinpoints to be created and wildcards that designate all (*) either in the parameter(s), names, or return type broaden the scope of joinpoint matching. As mentioned in section 3, Android has restricted native database access to very few libraries, and as such we avoided using wildcards where necessary and used more specific signatures instead.

In this test, we measured the time it takes to perform bytecode weaving as well as the number of joinpoints that are created. The bytecode weaving involves class parsing, joinpoint matching and insertion of advices for every class on the jar path as specified by the weaving aspect. On average it takes 15 seconds weaving time on our test platform to process each sample app, with the highest and lowest being 65.5 and 3.3 seconds respectively. The plot in figure 9 showed no correlation between instrumentation time and the number of joinpoints created. Nevertheless, we can see from the cluster that almost all the apps are weaved in less than 30 seconds. Manual investigation into the packages of the outlier applications indicates they contain a very large number of classes, thus requiring more time for the compiler to parse and match the joinpoints. Overall, we find the instrumentation time to be very acceptable as the maximum is slightly above 60 seconds.

In AspectJ weaving, for every matched joinpoint, the compiler adds a call to its corresponding advice. This is in addition to any Aspect-specific and Java-based method attached to the aspect class. Based on our sample set, we recorded an average of 1032 joinpoints created, with the highest being 7407 and the lowest 199. We find our joinpoint's overhead of approximately 1000 is on the high side considering there are about 16 advices. On investigation we find the pointcut that gets *application context* is the culprit. Though not part of the main functionality of our aspect, this pointcut serves as a helper that assigns context into the aspect's global context variable.

The Aspect class in not part of the traditional Android API and thus cannot instantiate a context. On the other hand, our advices requires it to get a ContentResolver for nested SQLstatements and the processing of ContentProviderOperations functions. Thus, we created this pointcut around the *onCreate* method of every Activity to get its context. This ensures at every point during the app's execution, a context is available to the aspect class. We find this method to be very reliable since even if one activity dies, the next activity will provide the needed context for the aspect, but not necessarily efficient.

6.3 Runtime Overhead

In this evaluation, we examined the impact of the introduced code on the app performance on the device at runtime. This is measured as the extra time it takes to run the advice on a joinpoint. Our advices verify access levels and enforce restrictions where necessary.

As mentioned in section 3, *priVy* provides three access levels, and when the access level is set to RESTRICT, zero or more schema, column and entity restrictions can be enforced. Thus, considering all this criteria, we expect different possible combinations for each of the CRUD function. It is important to note that this runtime overhead remains the same irrespective of the application executing because the advice code does not change. However, it is only affected by the access level and constraints enforced by the user. The more constraints there are, the more instructions are traversed and the greater the size of the SQLstatement.

To make this experiment possible we develop a testing app that triggers all our joinpoints and we run it many times with different combinations as shown in Table 2.

Each point on the table represents the average time in nanoseconds (ns) required to execute each CRUD function based on at least one restriction.

ALL_BLOCK and ALL_ALLOW incurs zero runtime overhead to process thus these are excluded from table. The times are computed by Java's *System.nanoTime()*. We set the start time at the beginning of the "around" advice execution and an end time at the beginning of its "after" advice. This ensures we take the time after the method has returned

Table 2: *priVy*'s Average Runtime Overhead Given Various Access Restrictions

Restrictions	Insert (ns)	Update (ns)	Query (ns)	Delete (ns)
1 Schema	59	40	64	47
1 Column	53	45	62	50
1 Entity	23	38	92	54
1 Schema , 1 Column	57	48	58	49
1 Schema , 1 Entity	76	47	70	69
1 Column , 1 Entity	74	50	73	67
1 Schema, 1 Column , 1 Entity	62	70	75	54
Average	57.71	48.29	70.57	55.71

and before the next instruction. The difference between the start and the end time is the runtime overhead per joinpoint.

Our experiments indicates it takes an average of 70.57 nanoseconds to execute the joinpoint on update, 55.71 for delete, 48.29 for insert, and 57.71 for query, with imposed restrictions. Overall our joinpoints take an average of 58.07 nanoseconds to execute any of their advice.

Since the static overhead gave us an average of 1032 joinpoints created, our instrumentation can incur a maximum runtime overhead of 0.06 milliseconds if all the joinpoints are executed in a single app. We find this overhead to be negligible and not be noticeable by users.

6.4 Access Policies

The access policies exposed by our Controller app enable users to protect the security and privacy of their devices and its data.

For instance, entity restriction can help protect devices from adding malicious contact on protected account types such as google.com or exchange. The entity restriction is set with an appropriate argument e.g., "com.google". Denial of service attacks due to malformed SQL can also be checked and special characters removed in the content values of an insert or update function.

Both Schema and Column restrictions can help reduce the exposure of clearly mapped data, e.g., enforcing schema restrictions on an email Uri can block a target app with access to Contacts from mapping contacts phone numbers and their email addresses. However, to attain absolute protection for some chosen contacts, the user can set up entity restrictions on individual names, IDs or groups. This will completely safeguard whole record(s).

Column restrictions can also protect permission leaks. As mentioned in Section 3, enumerating account names and types can give apps access to all accounts on the system just like those provided by *getAccount* permission. Thus, a user can set a policy to restrict these columns. This policy can further be optimized by enforcing entity restriction such that apps can access only their account name and types.

6.5 Limitation

priVy performs instrumentation via application repackaging and thus depends on the fact that the app will be correctly translated before and after instrumentation. However this may not be true for apps with:

1. Anti-repackaging techniques that detect and crash the translation process, often detected at compilation time. To deal with such problems, we use a well documented and widely used open source utility *dex2jar*. So far we have not encountered this issue, but it remains a possibility.

2. Signature verification that detects changes in the developer's original signature at runtime. Such an app may not necessarily crash but will fail to either render its activity or connect to its server. We have encountered some of these apps, which are mostly banking applications. Support for such applications is outside the scope of our research.

7. LITERATURE REVIEW

7.1 Android SQLite

The sensitivity of data and the disastrous effect of its breach has led RDBMSs to evolve over the years, incorporating different levels of security granularity at schemas, column and entity level. SE-PostgreSQL [15] for instance integrates PostgreSQL with SELinux such that every database object has a security context indicating its privilege and attributes. Oracle's virtual private database [10] and INGRES [22] on the other hand support fine grained access through runtime query modification. SQLite is an RDBMS which by design is a server-less jumbo file attached to an application. Its security layer is completely provided by the Android OS through the READ/WRITE permission system on the file. Although the file is protected from unauthorized access, privileges on the individual objects (schema, column, entities) are not segregated. SE-SQLite [16] like SEPostgreSQL is developed by integrating SELinux into Android SQLite. It provides low-level access control on database schema and tuples. Our work, though very much related in objective, differs completely in implementation. Theirs integrated the access policies on the database engine, while we enforced the access constraints at the application level by hooking the CRUD method calls and forcing query-rewriting where necessary. Our system does not require flashing a customized ROM, thus it is a more accessible and easily deployable solution.

7.2 Instrumentation

Instrumentation has been a vital tool for enforcing secure policies on Android systems both in static and dynamic contexts. Largely due to ease of application repackaging, static bytecode weaving at the application level has gained a lot of significance. Dr. Android [13] retrofits Android permissions using bytecode instrumentation. Capper [25] tracks sensitive information flow from source to sink while RetroSkeleton [8] enforces various flexible security policies at runtime. Appguard [4] and [5] both provides customizable user-policies through on-the-device application repackaging.

Most of these solutions have the same aim of reducing permissions associated with an Android app in general, whereas *priVy* is very specific to access control on SQLite database and its objects of which permission control alone cannot achieve.

Dynamically, FireDroid [19] and NJAS [7] use ptrace to attach their policy monitor to the target process. In both of these solutions, security policies are defined at a lower level by re-mapping the system calls to higher level API calls. Android PIN project also supports dynamic binary instrumentation. TISSA [26], Aurasium [24] Apex [17], all developed different security policies mostly with respect to reducing permissions by extending the Android framework. COMPAC [23] segregates permissions within the components of an application. AdDroid [18] segregates advertisement and the Android framework by introducing new advertisement APIs and permissions while AdSplit [20] executes the advertisement code in a different process.

ASM [11], SEAndroid [21], MockDroid [6] and AppFence [12] are operating system-centric solutions that developed integrated security policies at the kernel and Dalvik code. While the security policies suggested above can be used to either allow or deny access to the database file, that cannot address the issue of access control on the database object.

8. CONCLUSION

In this paper we presented *priVy*, a user-centric approach to enforcing object level privilege on Android native providers. Currently, database objects are not treated differently from their main source, meaning when access is granted to a SQLite database file, that access extends to all the objects encapsulated within it. The native databases contains enormously important data that should not be lumped together as a single entity, hence our motivation to segregate their access control. Our system *priVy* is designed to guarantee a user's privacy is secured in an accessible and highly usable way. It does not require operating system extensions nor does it tamper with the framework code, making it a much more practical solution than its contemporaries like SE-SQLite.

priVy leverages static bytecode instrumentation to weave in controlling code in database CRUD functions. The controller stub ensures only user approved schema, column, and/or entities are accessed by an instrumented application. When these CRUD operations are intercepted, the attached stub performs access level verification, query-rewriting where necessary, and proceeds with the function execution. It also performs database auditing when the attached app accesses any of the encapsulated objects. Our evaluation results demonstrated *priVy* incurs a minimal overhead of 15 seconds instrumentation time and a very negligible execution time overhead.

ACKNOWLEDGMENT

This work was partially funded by the NSF grant, CNS #1409534.

9. REFERENCES

[1] The aspectjtm development environment guide.

[2] Google play.

[3] pxb1988/dex2jar.

[4] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard–enforcing user requirements on android apps. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 543–548.

[5] Bartel, A., Klein, J., Monperrus, M., Allix, K., and Le Traon, Y. Improving privacy on android smartphones through in-vivo bytecode instrumentation. Tech. rep., uni. lu, 2012.

[6] Beresford, A. R., Rice, A., Skehin, N., and Sohan, R. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2011), HotMobile '11, ACM, pp. 49–54.

[7] Bianchi, A., Fratantonio, Y., Kruegel, C., and Vigna, G. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2015), SPSM '15, ACM, pp. 27–38.

[8] Davis, B., and Chen, H. Retroskeleton: Retrofitting android apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2013), MobiSys '13, ACM, pp. 181–192.

[9] Ferreira, D., Kostakos, V., Beresford, A. R., Lindqvist, J., and Dey, A. K. Securacy: An empirical investigation of android applications' network usage, privacy and security. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2015), WiSec '15, ACM, pp. 11:1–11:11.

[10] Heimann, J., and Needham, P. White paper :the virtual private database in oracle9ir2 - understanding oracle9i security for service providers. Tech. rep., Oracle Corporation, 2002.

[11] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. Asm: A programmable interface for extending android security. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 1005–1019.

[12] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 639–652.

[13] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 3–14.

[14] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. Getting started with aspectj. *Commun. ACM 44*, 10 (Oct. 2001), 59–65.

[15] Kohei, K. Security enhanced postgresql, 2013.

[16] Mutti, S., Bacis, E., and Paraboschi, S. Sesqlite: Security enhanced sqlite: Mandatory access control for android databases. In *Proceedings of the 31st Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC 2015, ACM, pp. 411–420.

[17] Nauman, M., Khan, S., and Zhang, X. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2010), ASIACCS '10, ACM, pp. 328–332.

[18] Pearce, P., Felt, A. P., Nunez, G., and Wagner, D. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2012), ASIACCS '12, ACM, pp. 71–72.

[19] Russello, G., Jimenez, A. B., Naderi, H., and van der Mark, W. Firedroid: Hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New York, NY, USA, 2013), ACSAC '13, ACM, pp. 319–328.

[20] Shekhar, S., Dietz, M., and Wallach, D. S. Adsplit: Separating smartphone advertising from applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 553–567.

[21] Smalley, S., and Craig, R. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS* (2013), vol. 310, pp. 20–38.

[22] Stonebraker, M., and Wong, E. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 annual conference-Volume 1* (1974), ACM, pp. 180–186.

[23] Wang, Y., Hariharan, S., Zhao, C., Liu, J., and Du, W. Compac: Enforce component-level access control in android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2014), CODASPY '14, ACM, pp. 25–36.

[24] Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 27–27.

[25] Zhang, M., and Yin, H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2014), ASIA CCS '14, ACM, pp. 259–270.

[26] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. W. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing* (Berlin, Heidelberg, 2011), TRUST'11, Springer-Verlag, pp. 93–107.