University of Toronto

Faculty of Applied Science & Engineering

Division of Engineering Science

ESC499 Thesis

Interim Report

Auditing File System on Android Specific Data

Yifan Wang

997036666

Supervised by

Professor David Lie

February 13, 2013

Table of Contents

List of Figures

List of Codes

1 Introduction

Mobile operating system security is becoming one of the most important topic in the computer security field at both the research and industry level. Specifically, due to its popularity and open source property, Googles Android OS (operating system) and associated software framework has been widely used as a way to judge security and efficiency without developing costly prototypes. Naturally, the theft prone property of mobile devices continuously challenges our ability to provide a sense of control over data ownership [1] [2]. To increase control, modern security design uses a combination of encryption and remote encryption key management that allow users to encrypt their file system while allowing fine-grain file access auditing and the ability to suspend future file access after compromising of device. Recent research into Android auditing file system such as Keypad [3] has demonstrated high potential for providing private data security. However, Android specific data types such as Contacts and Emails, are stored as entries in local SQLite databases as a single database file, and access is provided through an extension of the ContentProvider class [4]. As a result, the database and can only be audited as a single file access. The use of a auditing file system has proven to be effective in providing users with an encrypted file system while allowing fine-grained file access auditing [3], and should be advanced further for providing security of Android specific data.

The proposed research tests the effectiveness of auditing file system technique as described in Keypad architecture on Android specific data in its local database. The research approaches this problem from the design side, focusing on the development of a general technique for encrypting SQLite database entry, performing queries on encrypted database, as well as remote database replication and key management. Design of an optimized query performance is considered as motivation for this project, where the last stage goals may include batching of queries on the remote side for performance enhancement. However, the general task of designing a secure and auditable database is outside of the scope of the project at this point.

The method which was, and will be, followed is objective-focused and linear. It consists of three major sequential design stages, with a potential fourth stage. The fourth stage consists of a wide variety of studies which would provide a more detailed and modifiable context for the design developed in stages 1 through 3.

1. A typical Android application that utilizes local database storage will be built in JAVA. The application will be used to investigate various ways Android provides access to database content, as well as finding a central point of interception that is

general to all application where encryption can be applied with minimal effort and modification of existing system.

2. The database used in Stage 1 will be duplicated and synchronized on a remote server on a per query base. This includes arbitrary database schema creation, as well as operations such as query, insert or delete. This will allow an audit ability on the remote side of the database entries that are accessed or modified.

3. The database synchronization technique in Stage 2 will be enhanced with AES encryption with CBC mode. The new technique will allow security and auditing ability on database entry access from the Android application. Optimization techniques will be investigated such as database connection sharing between thread on the remote server, as well as batch query execution when queries arrive at a high frequency.

4. The scheme from Stage 3 will be put through a wide range of studies in order to study its efficiency under real workloads. Different encryption technique such as per-row versus per-column database entry encryption will also be investigated to observe the efficiency.

In successfully accomplishing the above four stages, the goals met will include the complete development and documentation of the auditing database system scheme as well as its application in the Android OS. The proposed research will give insight to allowing audit capability on general database system located in mobile operating system.

2 Literature Review

With the emergence of mobile device and mobile computing power, technologies offer us previously unimaginable global access to data, their theft prone property concurrently threatens our sense of control over data ownership. To protect the data stored on a mobile device in the case of loss or theft, today's users rely on traditional encrypted file systems, such as BitLocker and TrueCrypt [5]. However, traditional encrypted file systems are often insufficient to protect mobile data for two reasons. First, such systems can and do fail in the real world for many reasons: users configure weak passwords or write their passwords on easily accessible sticky notes, decryption keys can be recovered from memory, and most of today's tamper resistant hardware can still be physically compromised [1]. Second, when traditional encryption fails, it fails silently. After a device is stolen, users have no way of knowing whether a thief has compromised their protections and accessed any sensitive files.

Such difficulties have led to the development of auditing file systems such as Keypad [3], which supports fine-grained file access auditing that allows users to obtain explicit evidence of whether any files have been accessed after a device's loss as well as the ability to disable future file accesses after realizing that a device is lost. Keypad achieves these properties by weaving together encryption and remote encryption key management. Now Android smartphones amassed a commanding 68.1 market share of all mobile devices shipped during the second quarter of 2012 and operates under the Apache 2.0 open source license, which makes it an ideal candidate for a test bed of extension of Keypad design [6].

The following subsections introduce a number of key Android Operating System and Encryption key management concepts which were studied during the literature review. The current state of the field is then described, putting in context the possible contributions of this project. Finally, the research question driving this thesis is stated and a number of paths are presented for moving research forward.

2.1 Key Concepts

Prior to making any significant contributions to the field of auditing file system in mobile operating system security, a number of important and relevant concepts must first be examined and understood. The order of these concepts follows closely with the steps outlined for this project in Section 1. The technique that is used in Android to provide access for Android specific data is first considered. The design used by Keypad for remote key management that will be incorporated in this design is then explained. Finally, the

ideas behind the encryption technique for database entries are outlined.

2.1.1 Content Provider Class and Implementation

Android specific data are personal information of the users handled by system applications. These system applications include: PackageManager, Calender, Contacts, Downloads, Bookmarks, CallLogs, Gmail, SMS, and User Dictionary. User data are managed and stored as database entries in SQLite database stored locally on the mobile device.

SQLite is an opensource, serverless, self-contained database implementation in C [7]. Android implemented SQLite in the Native layer as part of its library, and on top of that, a Java wrapper layer in its application framework. Android provided serval ways for an application to interact with this database as outlined in Table 1.

Table 1: Methods for Android Application to Interact with SQLite Database

Method	Description
SQLiteHelper	Create *SQLiteDatabase* -> Create an *ApplicationDatabaseHelper* which extends *SQLiteOpenHelper* -> Call super class constructor with database name -> Interact with the database with *SQLiteDatabase.execSQL()*
ContentProvider	Designing Content URIs -> Implementing the ContentProvider Class -> Query the *ContentProvider* through *ContentResolver* Class

If the database is completely local, application will employ *SQLiteHelper* method. This class does the heavy lifting of creating and interacting with the database for the application, while the application can execute arbitrary queries through the *SQLiteDatabase.execSQL()* method.

Applications that needs to store complex database schema and would also like to share its database content with other applications, interact with the SQLite database through an application framework *ContentProvider*. It is a primary building block of Android application, and have been part of the Android system since API level 1.

ContentProvider provides a series of abstract methods to be implemented by developers to allow flexibility and customization for each application: query(); insert(); update(); delete(); getType(); onCreate().

To better understand this concept, Figure 1 demonstrates the hierarchical relationship between application and its provider class. In the Contact application, ContactProvider

implements ContentProvider class to provide specific ways the Contact application wants to interact with the database that stores contact information. Contact application interacts with ContentProvider through ContentResolver, which takes an Uri object that maps query request to objects in the database.

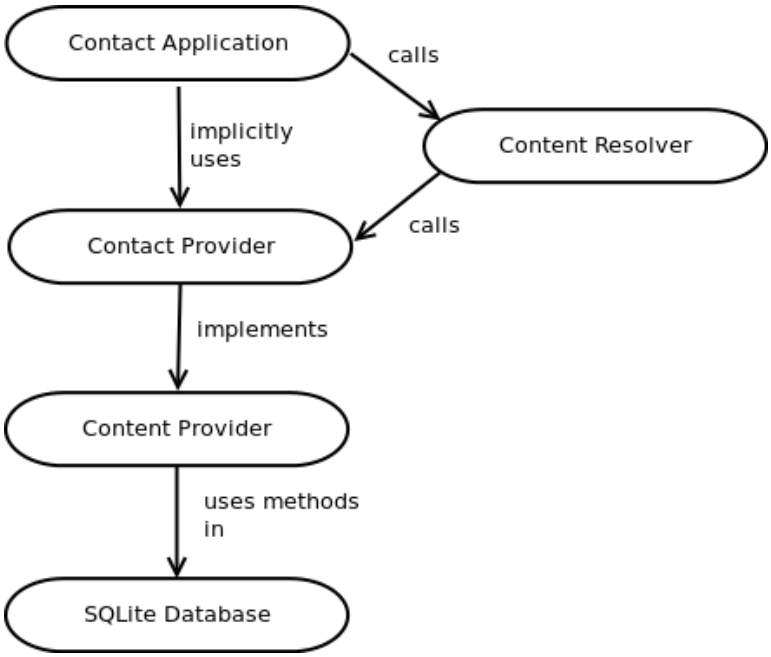

Figure 1: Provider hierarchy for Contact Application

2.1.2 Auditing File System for Android Specific Data

Auditing File System is a method of storing encryption key on a remote server for local data that is encrypted [8]. The advantage of this method is that each time a local file is accessed, the encryption key is fetched from the remote server, thus can be logged. Also, if the local data storage is compromised, key access can be disabled remotely thus preventing all future access.

At time of data creation, the data is encrypted, and the key is sent to a remote audit service. As out lined in Figure 2.1.2, when client request access to a piece of data, the key is then fetched from the server. Each piece of local data is encrypted with its own symmetric key. Client downloads the key for a file each time it is accessed, and destroys the key immediately after use.

To properly implement this method, several area needs to be well thought out. First of all, to manage the key stored on remote side efficiently, a database schema specific for the

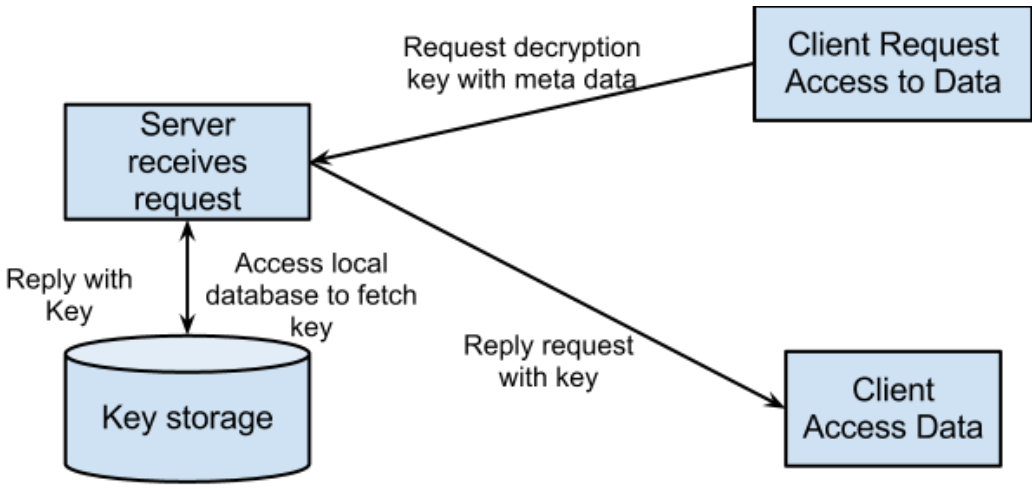

Figure 2: Request map for Remote Encryption Key Storage and Management

application for key storage needs to be implemented. Second of all, the request should contain minimal information such that the network usage is minimized. Third of all, server should be able to handle concurrent requests for concurrent data access. These aspects will be properly addressed in Section 3.

2.1.3 Per-row vs Per-column encryption

To encrypt entries of database, and remotely manage its keys, we need to explore the benefit of different encryption schemes. As will be shown, per-row encryption allows fine-grain auditability but requires complete duplication of unencrypted database on the server side, while per-column encryption allow coarse-grain auditability, but only requires duplication of database schema on the server side. For each method, we will look at how database query will be implemented on a sample contact application.

Per-row encryption This method encrypts each row with a unique encryption key. The client side database will have each row encrypted with an unique encryption ID. Since querying an encrypted database directly is impossible, a copy of the unencrypted database is maintained on the server side with encryption id as well as decryption key. The sample database schema is outlined below.

Server: (EncID, Name, Email, Age, Key)

Client: (EncID, Name, Email, Age)

As shown in Figure 3 Insert will be performed twice, once with unencrypted data on the server side, and second time with encrypted data on the device side.

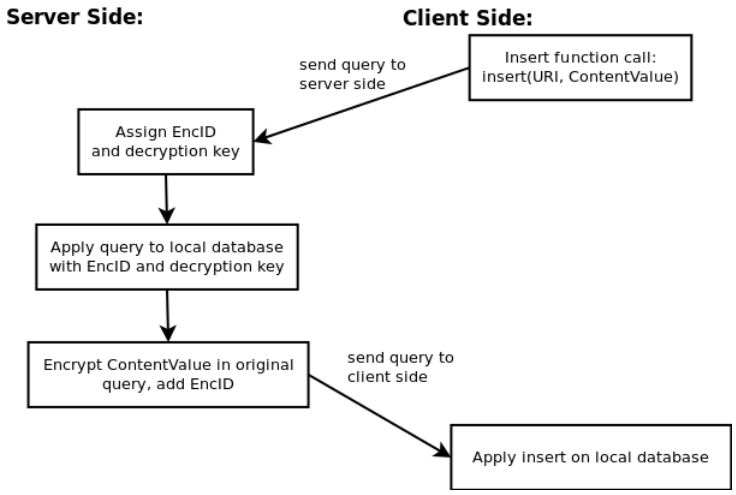

Figure 3: Requset map of Insert method call for Per-Row encryption method

As shown in Figure 4, two different type of query will be implemented with different logic. When the query involves a simple projection, a similar project on the column of decryption key is replied. Projection with selection requires us to find the specific EncID first, then send it back to the client side for the data to be queried and decrypted.

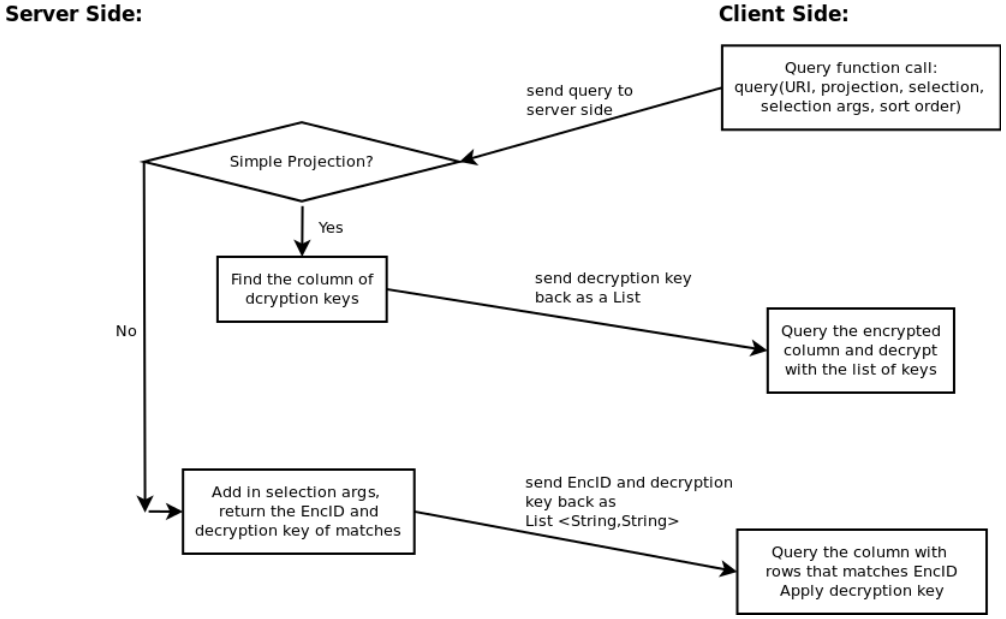

Figure 4: Requset map of Query method call for Per-Row encryption method

Per-column encryption This method encrypts each column with an unique encryption key. The client side database will have each column encrypted. The server side database will simply maintain a list of decryption key for each column.

Server: (Column, Name, Key)

Client: (Name, Email, Age)

Since every column uses the same encryption key we simply need to request the key maintained on the server side and encrypt the data entries inserted into local database as outline in Figure 5.

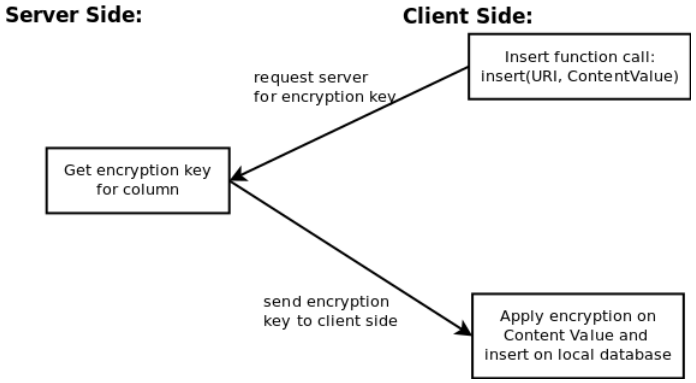

Figure 5: Requset map of Insert method call for Per-Column encryption method

Since there is no way for us to query on encrypted data, we will decrypt the entire column, and perform query as outline in Figure 6. After the query is completed we re-encrypt the entire column. This is similar to having an decryption key expired on a documented that is encrypted.

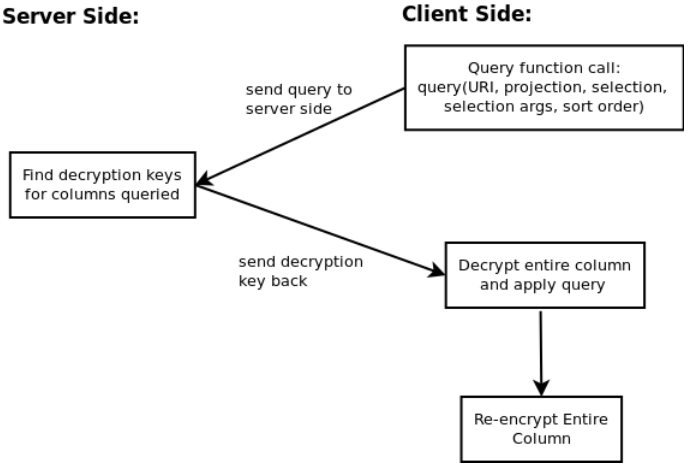

Figure 6: Requset map of Query method call for Per-Column encryption method

2.2 Current State of Field

To properly understand the current state of field relavent to this research, we first realize that the method employed is a combination of database encryption and remote key management.

Database encryption is a well explored topic. *CryptDB* for example, is an open source database management system (DBMS) that operates on an encrypted database. It associates user password with the queries they performed directly on encrypted data. As a result, user can only view data relavent to themselves [9]. This solves the problem when a database administrator may want to access private data such as health record, as well as the situation when the entire DBMS is compromised. It works by realizing the idea that all SQL queries are made up of a set of well defined operators such as equals, orders and joins. It uses known encryption schems and privacy-preserving cryptographic method to allow DBMS to execute on encrypted data. Other open source database encryption methods also exists. Specifically for SQLite on Android, *SQLCipher* is an extention to SQLite database platform that allows creation of encrypted database [10]. SQLCipher uses 256-bit AER in CBC mode for encryption, and once encrypted, queries performed on encrypted data will be unresolvable. Due to its low overhead and compact size, it is quickly becoming the most heavily used encryption database solution for mobile developers.

The majority of research in the field of auditing file system and remote key manamgent is very recent. Specifically as mentioned in Section 1, *Keypad* is an encrypted file system that aims to provide autiablitiy on data on the mobile device when the device is lost of stolen. The most obvious challenge for remote key management is posed by network availability. Keypad presents two unique methods to solve performance issue over slow or high-latency networks. First of all, encryption keys are cached and prefetched. For key caching, a experimental result of 100 seconds were determined to be most efficient [3], key prefetching is performed for example, when a recursive file operation is detected such as searching. As for disconnected access, Keypad proposed to use paired devices over conntection such as bluetooth for accessibility. The most recent work by Professor David Lie at the University of Toronto has taken a similar approach as Keypad, and applied it to Android. The design extends EncFS [11] byadding a unique file ID to the header of files. This ID is requested by the client at time of file creation and generated by the remote server. Similarly, when the file is accessed, the key associated with the same file ID is requested, and is expired after the file is closed. This appoarch enables auditability on Android powered device.

Although there is a strong literature background for the methods described in Sections 2.1.1 to 2.1.3, the combination of these methods has not been explored yet. For example, neither *CryptDB* and *SQLCipher* allows the tracking of individual encryption key for each data entry, thus lacking auditing ability. At this point, an auditable database system for Android specific data has yet to be developed. With the inclusion of database entry encryption and remote key management, the design has potential to be very effective at providing sigificantly higher degree of security and ownership over private data.

2.3 Research Question and Study Paths

The major research question being tackled is: *Investigate if Auditing File System protection can be extended to Android-specific data types, such as contacts, calendar items, browser bookmarks, and investigate the overhead and implementation effort of such protection.* As mentioned in Stage 1 in Section 1, the question of where is a common point where client side query on the database can be intercepted needs to be answered. Following that, studies focusing on performance need to be conducted in order to effectively judge the effectiveness of this method. Once the design is developed, the field of research can be moved forward along a number of paths.

2.3.1 Choice of Model

2.3.2 Performance Overhead

2.3.3 Encryption Method and Optimization

3 Progress to Date

At this point, the progress to date includes a complete sample Contact Application for design benchmarking, a completed investigation of finding a central point of interception that is general to all application where encryption can be applied with minimal effort and modification of existing system, a complete database synchronization between client and server, and significant progress in the implementation of the transferring database query from client to server. When referring to the stages outlined in Section 1, the current state of the project is approximated to be in the beginning of Stage 3. The following subsections describe the methods used in, and results obtained from, developing the server and client side logic.

3.1 Sample Contacts Application and Intercepting Queries

The method used to develop the Sample Contacts application is fairly straightforward. In order to understand where all the query can be intercepted, we first need to look at how the application extends a ContentProvider. The application consist of three main components: a ListViewActivity displaying a contact for each row; a detail Activity displaying the detail of each contact; and a ContentProvider that provides access to contacts stored in the SQLite database.

First, the database and data model needs to be determined. Code 1 presents the database schema used in the sample application.

Code 1: Java implementation of Database Schema for Sample Contact Application

```
public class ContactTable {

    // Database table
    public static final String TABLE_CONTACTS = "contacts";
    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_NAME = "name";
    public static final String COLUMN_EMAIL = "email";
    public static final String COLUMN_AGE = "age";

    // Database creation SQL statement
    public static final String DATABASE_CREATE = "create table "
            + TABLE_CONTACTS
            + "(" + COLUMN_ID + " integer primary key autoincrement, "
            + COLUMN_NAME + " text not null, "
            + COLUMN_EMAIL + " text not null,"
            + COLUMN_AGE + " integer not null" + ");";
```

```
    public static void onCreate(SQLiteDatabase database) {
        database.execSQL(DATABASE_CREATE);
    }
}
```

A ContactDatabaseHelper class was implemented that extents SQLiteHelper. In the onCreate method of this class, a database is created with the schema outlined in ContactTable.DATABASECREATE statement. Finally, a ContactProvider class is implemented to extend ContentProvider as shown in Code 2 that provides methods such as query and insert. In the ContactProvider class, a URIMatcher is created as a standard helper method for helping the provider figure out which data the query is intended for, for example, a single contact, or a set of contacts. The ContactProvider is called via the ContentResolver class, such as getContentResolver().insert().

Code 2: Sample Contacts Application own implementation of ContentProvider

```
public class ContactProvider extends ContentProvider {
    // database
    private ContactDatabaseHelper database;

    // Used for the UriMacher
    private static final int CONTACTS = 10;
    private static final int CONTACT_ID = 20;

    // Content URI
    private static final String AUTHORITY =
       "se.yifan.android.encprovider.SampleContacts.contentprovider";
    private static final String BASE_PATH = "contacts";
    public static final Uri CONTENT_URI =
       Uri.parse("content://" + AUTHORITY + "/" + BASE_PATH);

    // MIME type for multiple rows
    public static final String CONTENT_TYPE =
       ContentResolver.CURSOR_DIR_BASE_TYPE + "/contacts";
    public static final String CONTENT_ITEM_TYPE =
       ContentResolver.CURSOR_ITEM_BASE_TYPE + "/contact";

    private static final UriMatcher sURIMatcher =
       new UriMatcher(UriMatcher.NO_MATCH);

    static {
        sURIMatcher.addURI(AUTHORITY, BASE_PATH, CONTACTS);
        sURIMatcher.addURI(AUTHORITY, BASE_PATH + "/#", CONTACT_ID);
    }

    @Override
    public boolean onCreate() {...}
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
```

```
       String[] selectionArgs, String sortOrder) {...}
    @Override
    public Uri insert(Uri uri, ContentValues values) {...}
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {...}
    @Override
    public int update(Uri uri, ContentValues values, String selection,
       String[] selectionArgs) {...}
}
```

To be able to intercept all queries performed by the application to the database, we need to act as a middleman between the application and ContentProvider. The method used here, is to create a EncProvider class that also extends ContentProvider, and have ContactProvider extend EncProvider, and call its super methods. As a result, each corresponding methods of ContentProvider will go through EncProvider and be rerouted to the server.

Code 3: Sample Contacts Application extending EncProvider

```
public class ContactProvider extends EncProvider {
    ...
    @Override
    public boolean onCreate() {
         super.onCreate();

    }
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
       String[] selectionArgs, String sortOrder) {
         super.query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder);

    }
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        super.insert(Uri uri, ContentValues values);

    }
     @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        super.delete(Uri uri, String selection, String[] selectionArgs);

    }
     @Override
    public int update(Uri uri, ContentValues values, String selection,
       String[] selectionArgs) {
        super.update(Uri uri, ContentValues values, String selection,
           String[] selectionArgs);

    }
```

```
}

public class EncProvider extends ContentProvider{
    ...
    @Override
    public boolean onCreate() {...}
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
       String[] selectionArgs, String sortOrder) {...}
    @Override
    public Uri insert(Uri uri, ContentValues values) {...}
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {...}
    @Override
    public int update(Uri uri, ContentValues values, String selection,
       String[] selectionArgs) {...}
}
```

3.2 Query forwarding and Database Synchronization

To allow transfer of query performed on client database to server database, the method employed here uses a socket server, and an own implementation of packet for data transfer.

The packet contains information needed to represent any particular type of query with the query content as shown in Code 4. For example, when a query is performed on the client, a QueryPacket is initialized as demonstrated in Code 5.

Code 4: Implementation of QueryPacket for transferring query information accorss client and server

```
public class QueryPacket implements Serializable {
    public static final int DB_NULL = 0;
    public static final int DB_CREATE = 100;
    public static final int DB_QUERY = 101;
    public static final int DB_INSERT = 102;
    public static final int DB_DELETE = 103;
    public static final int DB_UPDATE = 104;

    public static final int ERROR_INVALID_SYMBOL = -101;
    public static final int ERROR_OUT_OF_RANGE = -102;
    public static final int ERROR_SYMBOL_EXISTS = -103;
    public static final int ERROR_INVALID_EXCHANGE = -104;

    public int type = QueryPacket.DB_NULL;

    public String db_name;
    public String table;

    public String db_creation;
```

```
    public String contentValues;
    public String nullColumnHack;
    public String whereClause;
    public String db_query;

    public String key;
    public String[] args;
    public int[] contentType;
}
```

Code 5: Initialization of QueryPacket for performing Query on server

```
toServer = new QueryPacket();
toServer.type = QueryPacket.DB_QUERY;
toServer.db_name = dbName;
toServer.db_query = sql;
toServer.args = selectionArgs;
```

The server creates a socket and listens to any incoming connection when it is first created. It uses SQLite database locally due to its lightweight property as well as no major performance difference as shown in table . The library used for interacting with SQLite database on the server side is sqlite-jdbc[8], this library is chosen mainly for two reasons: first of all, it allows for an easy transition to any other jdbc enabled database if needed; second of all, it allows argument binding for queries over other Java implementations[9]. Once a connection is received, any incoming connection will create a new thread to allow concurrent query from client. In each thread, a state machine determines the type of incoming packet and performs needed query on the local database accordingly. After the action is performed, a reply packet is then sent to client containing information such as encryption keys.

[insert Table]

Query handling could be very tricky since different type comes with different data wrapped in Query packet. As a result, the server side logic can become very complicated. For example, build a SQL statement for insertion involves taking care of different data types. As shown in Code 7, a bindArgument method is used to build SQL statement for insert, while in Code 6, a buildSQLStatement method is used to build SQL statement for query and delete.

Code 6: Method that builds a SQL statement given a set of arguments the statement needs

```
private PreparedStatement buildQuerySql(String sql, String[] sqlArgs)
   throws SQLException {
        PreparedStatement preparedStatement = connection.prepareStatement(sql);
```

```
        if (sqlArgs != null) {
            int i = 0;
            for (String args : sqlArgs) {
                preparedStatement.setObject(i++, args);
            }
        }
        return preparedStatement;
    }
```

Code 7: Method that builds a SQL statement given the argument and argument type

```
private StringBuilder bindArguments(StringBuilder sql, Object[] bindArgs,
   int[] bindArgsType) {
        final int count = bindArgs != null ? bindArgs.length : 0;

        if (count == 0) {
            return sql;
        }

        for (int i = 0; i < count; i++) {
            final Object arg = bindArgs[i];
            switch (bindArgsType[i]) {
                case Cursor.FIELD_TYPE_NULL:
                    break;
                case Cursor.FIELD_TYPE_INTEGER:
                    sql.append((i > 0) ? "," + ((Number) arg).longValue() :
                     ((Number) arg).longValue());
                    break;
                case Cursor.FIELD_TYPE_STRING:
                default:
                    sql.append((i > 0) ? ", \"" + arg.toString() + "\"" :
                     "\"" + arg.toString() + "\"");
                    break;
            }
        }
        return sql;
    }
```

4 Future Work

As mentioned in Sections 2 and 3, a large amount of research and development has already been conducted. Nonetheless, a number of key stages still remain. This section outlines the remaining stages and goes into some detail to describe possible methods and expected outcomes.

4.1 Encrypted Database Entry

The development of encryption per database entry is probably the most crucial stage of the project. As outlined in Section 2.1.3, the most algorithm is largely simple to implement. The encryption algorithm will use is standard AES with CBC mode.

A key decision at this point is determining the placement encryption on client or server side. The advantage of applying encryption on server side is more computational resources, but this also means that encrypted content along with encryption key need to be sent back to the client one encryption is complete, and could become expensive when data inserted to the database is large. If encryption is applied at the client side, then the encrypted data can be written into client database without having to go through server, thus saving network bandwidth.

4.2 Possible Studies

A number of different numerical studies can be conducted. As mentioned in Section 2.3, the ideal outcome of these studies is a set of data that will help assess the efficiency of auditable database design.

The most basic study would be a comparison of overhead produced by added layer of EncProvider (Section 3.1). A set of real-world workload on a contact database can be recorded through tracing database query calls on the Contact database as part of the Android operating system. These queries can then be played back on the benchmark application with and without EncProvider. A wall clock time will be recorded to study the performance overhead.

A second study that will be performed is a comparison between per-row and per-column encryption as outlined in Section 2.3.2. A set predetermined queries will be played back at random time interval with each encryption method as outlined in table.

[insert Table]

4.3 Schedule

The Gantt chart in Figure 7 outlines the schedule from now until the end of the term when the final thesis document is to be delivered.

References

[1] J. Halderman and S. Schoen, "Lest we remember: Cold boot attacks on encryption keys," *In Proceedings of the 17th USENIX Security Symposium*, 2008.

[2] T. M. D. Dagon and T. Starner, "Mobile phones as computing devices: the viruses are coming!," *Pervasive Computing, IEEE*, 2004.

[3] R. Geambasu, "Keypad: An auditing file system for theft-prone devices," *EuroSys'11 - Proceedings of the EuroSys 2011 Conference*, 2011.

[4] Google, "Contacts provider, android component manage repository of people." `http://developer.android.com/guide/topics/providers/contacts-provider.html`, 2008.

[5] M. Blaze, "A cryptographic le system for unix," *In Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS93)*, 1993.

[6] I. D. Corporation, "Worldwide quarterly mobile phone tracker." `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37#.URvB83zCD3w`, 2012.

[7] D. R. Hipp, "Sqlite relational database management system." `http://www.sqlite.org`, June 2000.

[8] F. Cohen, "Information system defences: A preliminary classification scheme," *Computers Security*, vol. 16, no. 2, pp. 94–114, 1997.

[9] N. Z. Raluca Ada Popa, Catherine M. S. Redfield and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," *In Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[10] Z. LLC, "Sqlcipher - an open source library that provides transparent, secure 256-bit aes encryption of sqlite database files." `http://sqlcipher.net`, 2008.

[11] EncFS, "Encfs encrypted filesystem." `http://www.arg0.net/encfs`, 2008.

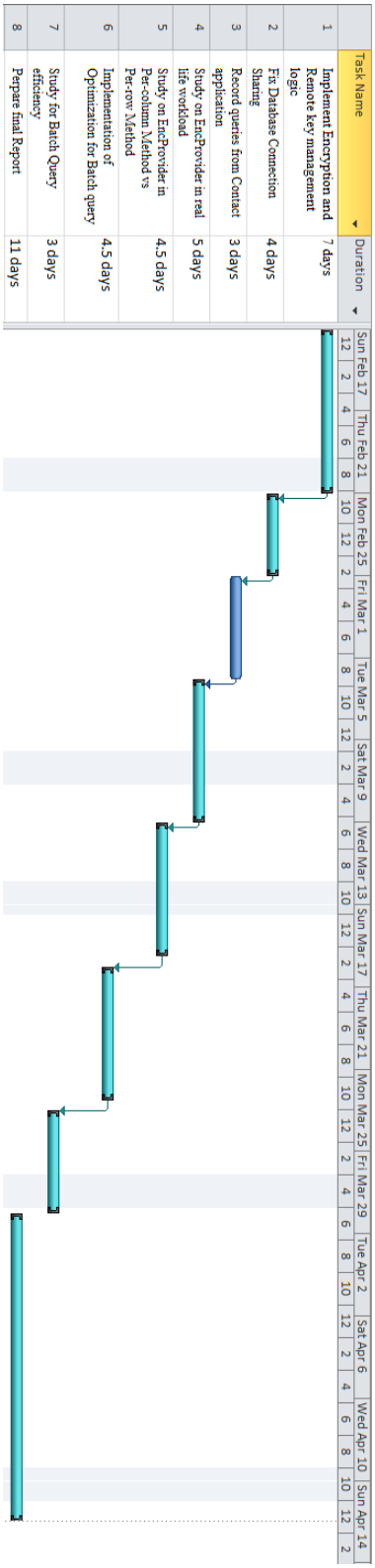

Figure 7: A Gantt Chart detailing the work to be done on this project in the future.