RESEARCH REPORT
OCCASIONAL PAPER

Attribute-Level Encryption of Data in Public Android Databases

Charles E. Loftis, Tennyson X. Chen, and Jonathan M. Cirella

September 2013

RTI Press

RTI INTERNATIONAL

About the Authors
The authors work in RTI International's Research Computing Division. Their main focus is the National Survey of Drug Use and Health (NSDUH) project. **Charles E. Loftis**, MS, research analyst, is key web, mobile, and database developer on the NSDUH project.

Tennyson X. Chen, MS, is a senior research analyst and software system architect. For the NSDUH project, he is a key system designer and database manager.

A software developer and database practitioner, **Jonathan M. Cirella**, BS, designs and maintains application databases for the NSDUH project.

RTI Press publication OP-0016-1309

Suggested Citation

Loftis, C. E., Chen, T. X., & Cirella, J. M. (2013). *Attribute-level encryption of data in public Android databases* (RTI Press publication OP-0016-1309). Research Triangle Park, NC: RTI Press.

This publication is part of the RTI Research Report series. Occasional Papers are scholarly essays on policy, methods, or other topics relevant to RTI areas of research or technical focus.

RTI International
3040 Cornwallis Road
PO Box 12194
Research Triangle Park, NC
27709-2194 USA

Tel: +1.919.541.6000
Fax: +1.919.541.5985
E-mail: rtipress@rti.org
Web site: www.rti.org

doi:10.3768/rtipress.2013.op.0016.1309 **www.rti.org/rtipress**

Attribute-Level Encryption of Data in Public Android Databases

Charles E. Loftis, Tennyson X. Chen, and
Jonathan M. Cirella

Abstract

Android mobile devices have become an attractive consumer product because of their portability, high-definition screens, long battery life, intuitive user interface, and ubiquitous competitive vendor pricing. The very feature that has helped with the proliferation of the devices is also one of the most problematic: their portability could result in theft, potentially allowing data to be compromised. For applications deployed to these devices, data security requirements need to be incorporated in the design process so these devices can be considered viable data collection tools. Researchers at RTI have been working to secure data on Android mobile devices so that selected information on the device can be encrypted and therefore difficult to obtain illegitimately while still making confidential data easy to access. We have developed software that will encrypt specific attributes of databases residing on the internal secure digital card (SD card) of Android devices. The method we have developed could also benefit other Android applications requiring secure storage of data on globally readable and writable databases. In this occasional paper, we discuss the technologies and methods used in our Android database encryption/decryption implementation and their potential scalability to broader applications.

Contents

RTI INTERNATIONAL

Introduction

Each year public, private, and government entities spend significant resources collecting information pertinent to their business practices. The information collected is often critical in making policy, budget, resource allocation, and purchasing decisions. Obtaining and securing data are therefore essential tasks for organizations relying on the information collected.

Over the last few years, devices running Google's Android operating system (OS) have become attractive to consumers and are increasingly used by organizations to collect data. Some reports show that Android has captured as much as one-half of the smartphone market.[1] However, as the popularity of these portable handheld devices increases, the risk that these devices may be lost or stolen also increases. This risk poses a significant concern to entities that need assurance that their data are securely stored and safe from damaging compromise.

The Android OS provides security by "sandboxing"[2] applications (apps). Each app is installed into its own isolated directory, and permissions are assigned such that no other app is allowed to access those resources.[3] Assuming that the OS and the associated mechanisms for enforcing file access permissions remain robust, this is an appealing security model. However, Android smartphones and tablets have historically been, and continue to be, relatively easy to "root." In other words, the OS can be quickly hacked, giving the hacker administrative, or root, privileges in Android and invalidating the built-in security model for apps and their associated data structures. Based on our observations and as noted by Pocatilu,[4] encryption is the only practical way to protect sensitive data given the ease with which devices can be rooted and security mechanisms defeated.

Android provides two primary methods for encrypting data: whole device encryption or file encryption. Recent versions of Android (version 3.0+) provide a full device encryption option. This option encrypts all data saved to internal, private storage. Data stored in the private encrypted area can only be decrypted by the OS. For example, a file extracted from the encrypted internal storage will be unreadable without decryption by the OS and should therefore be safe from compromise. However, a presentation at the DEF CON 2012 conference outlined a method for defeating Android's whole device encryption.[5]

The second way data can be encrypted on Android, regardless of where that data may reside (i.e., private file system or public secure digital [SD] card memory), is to encrypt the target file itself. This option requires the development or installation of third-party solutions or algorithms. The devised solution must be implemented, applied, and maintained when updates (i.e., to third-party solutions) are released.

Motivation

Data Collection

The research described in this paper was motivated by work RTI International conducts on several large nationwide field surveys on an ongoing basis. These surveys typically gather responses to questions of interest to social science researchers, related to subjects such as education, drug use and abuse, mental health, and sexually transmitted diseases and associated risk behaviors. At any one time, several surveys may be active in all 50 states, as well as internationally, with operation around the clock, 365 days per year. RTI collects data from hundreds of thousands of survey respondents annually. These data are first collected and stored in Android devices for future transmission back to RTI servers. Since the databases underlying our survey apps often contain sensitive information, confidentiality and security are important considerations.

Data Transmission

Project protocols typically require that data collected by interviewers be communicated back to RTI as soon as possible using local resources. Interviewers often collect data in rural and underdeveloped areas where wireless Internet (Wi-Fi) connections and high-speed Internet connections may not be available. Quite often our interviewers have access to the Internet only via dial-up modem connections, and Android devices do not directly support this communication mode.

Because Wi-Fi Internet access is not available to all interviewers, they must tether the Android device via a Universal Serial Bus (USB) cable to a project-provided laptop and use the shared dial-up modem connection from the laptop to transmit the data via analog modem.

Enabling data transmission via a tethered laptop provides multiple ways for in-field interviewers to send data to home servers via various Internet connections: dial-up, wireless, Ethernet, and 3G/4G. The laptop is used to broker the communication, so we can use any mode supported by the laptop hardware and software.

Implementing the tethered data transmission system requires that the database on the Android device storing survey data be publicly (in terms of the Android OS) accessible. Therefore, in this paper we concentrate on the problem of encrypting SQLite[6] databases on an Android device's globally readable and writeable internal storage secure digital (SD) card. The same technology can easily be adopted for other database formats.

SQLite was developed as an open source project to store data in flat file database storage on the device. It confirms to the transactional Structured Query Language (SQL) standard, without the administration complexity of hosting a SQL server application instance. It is commonly used in the software development industry to store data in non-volatile memory data format.

Problem

Recent versions of Android provide an option for encrypting files in internal storage; this includes data stored in the root /data directory. Each app installed on a device is provided its own subdirectory under /data where it can store information. By default, the entire /data directory is protected by the Android OS and is not readable by external applications and programs. However, in many situations it is convenient, if not absolutely necessary, to store app data in a location that can be accessed by multiple apps, as well as by a USB-connected host. The data transmission scenario described above is a prime example. Because our data are stored in a public storage area unprotected by the Android OS, encryption of sensitive data is critical.

Android provides full native support for SQLite databases[7] but does not provide encryption support. A few notable extensions providing encryption support to SQLite exist: SQLite Encryption Extension[8] and SQLCipher[9] are two of the most popular. Both of these packages require a developer to acquire source code and build libraries and/or pay a fee for compiled libraries and/or pay for support in executing the compilation. In the end, both packages are initially free but require time and resources to learn application programming interfaces (APIs) and interact with third-party support staff. Further, annual support costs, depending on the level of service needed, range from \$899[10] to \$75,000[11] and can burden project budgets. Practically speaking, the need to pay annual support fees, plus the effort required to integrate a third-party solution, can prove to be cost prohibitive for small to medium app development projects. Therefore, we sought a more generic approach to encrypt our survey data. Our approaches and methods are discussed in detail in the following sections.

Solution Evaluation

Concluding that the existing full-device encryption method and the third-party encryption libraries do not suit our needs in protecting sensitive data while providing flexible data access within budget constraints, we explored the possibility of implementing our own data encryption scheme and developed a method for database attribute-level encryption.

We offer a brief overview of the full-device encryption architecture provided by later versions of the Google Android mobile OS to highlight its advantages and disadvantages. The semantics of the encryption framework are based on dm-crypt (a Linux kernel feature) and Android volume daemon (vold) invoking encryption commands at the file system level. Initial versions of the encryption are based on 128-bit Advanced Encryption Standard (AES) with cipher block chaining, encrypted salt-sector initialization vector, and the Secure Hash Algorithm using a master key encrypted with 128-bit AES created from the OpenSSL library.[12]

To use these strong encryption features, the mobile device must have a lock screen to restrict access by password, personal identification number, or gesture or by using device owner facial recognition. The lock screen restriction is used as a seed value for the encryption modules. The Android OS level of encryption framework provides adequate security, preventing most unauthorized access if the device is lost or stolen. The lock screen password seed value is necessary to decrypt the data stored on the device. In addition, a certain number of failed password attempts can be specified such that once reached, all sensitive data are wiped off of the device.

Drawbacks to Android's full-device encryption include the following: (1) the public internal memory (i.e., SD card) does not get encrypted, (2) older versions of the Android OS do not offer the full-device encryption feature, and (3) communication between the Android tablet and a USB-connected host can become clumsy and difficult to implement. Further, when the lock screen timeout period is set to be too long (e.g., 30 minutes), the effectiveness of the entire mechanism is compromised. A lost device with an unlocked screen exposes potentially sensitive data. This is hard to work around, because the screen timeout is configurable by the user, and restrictions on those settings are difficult or impossible to impose.

We also considered using third-party cryptographic packages that use their own implementation of SQLite encryption or proprietary database packages. The main benefit of a third-party package is that it abstracts the complexity of encryption away from the programmer, reducing the amount of programming development. However, third-party packages may incur consulting costs that could burden project budgets, and the development effort relies on the availability of third-party support, which is especially important when software problems occur. A desire to minimize cost, coupled with a need to maximize flexibility by minimizing the use of third-party solutions, led us to search for other options. In particular, we decided to examine methods that would provide attribute-level encryption to obfuscate specific database attributes by converting them from plain text to encrypted, scrambled cipher text.

Our apps need to have tight control of the symmetric key seed value because we also need to match the Android app encryption with corresponding server-side data decryption procedures. We observed that it is faster to encrypt a small number of sensitive attributes than to encrypt a whole database.

The attribute-level encryption method provides several advantages. First, it provides strong encryption that adequately satisfies Federal Information Processing Standards (FIPS) publication 140 risk[13] data security requirements. Depending on the cryptographic package used, the attribute-level encryption method can be tailored to meet FIPS 140 low (Level 1) or moderate (Level 2) data security levels. Android's native crypto package meets FIPS 140 low data security risk requirements. Second, because we are developing the encryption classes/methods, we control the source code and can adjust it according to specific project security requirements. Similarly, because we control the code, maintenance and issue discovery/remediation is simpler than with a third-party solution. Finally, this method does not require license or support purchases, making it attractive for long-running projects with dedicated support staff (e.g., projects with dedicated staff but not direct cost budget overhead). Because of these advantages, we decided to adopt attribute-level encryption for use in survey apps requiring strong protection for databases stored in shared memory areas within Android devices. In the next section, we describe our chosen approach in more detail.

Proposed Approach

We first discuss the testing environment we used to evaluate our attribute-level encryption method. The environment consists of an Android device running a locally installed native Android app, collecting data and storing it into a SQLite database located in the internal memory of the device's SD card. A separate Windows-based (x86) data transmission program installed on a laptop interfaces with the SQLite database via a USB connection to the Android device. The following steps are required for the x86 data transmission program to proceed. First, the

SQLite database file is retrieved from the Android device using the Android Debug Bridge (ADB) pull command. The database is stored in a temporary location on the laptop's hard drive memory. Next, the x86 data transmission program synchronizes the temporary database on the laptop to the server database using RESTful web services and HTTPS communication. Finally, the x86 data transmission program pushes the temporary SQLite database back to the Android device from the laptop using the ADB push command. Our testing environment is shown in Figure 1.

To prevent identification of any individual or location being surveyed, we decided to encrypt any attribute containing personally identifiable information (PII): names, addresses, and phone numbers. Our analysis revealed that a relatively small amount of these types of personally identifiable attributes (less than 3 percent of the 200 or so attributes stored in each project database) needed to be protected. Therefore, we felt that encrypting names, addresses, and phone numbers at the database attribute level was an ideal solution.

Our approach requires the user to enter a shared symmetric key passphrase and the app to create or recover a private key (for each attribute value) that is embedded in the final cipher string. Figure 2 demonstrates the encryption process. To encrypt a clear text string, we first obtain a private key by generating a random 13-digit number. Using the Secure Hash Algorithm (SHA2), we encode the private key. The Android device uses the hashed private key as the initialization vector (IV) along with the public symmetric key passphrase and clear text as inputs into the 256-bit AES encryption algorithm. It then converts the resulting cipher text to a hexadecimal value, prefixed with the unhashed, hex-encoded private key. The scrambled cipher text is then stored by the app in the SQLite database residing on the Android device's internal public SD card for future access. To decrypt the cipher text, the app parses the encrypted text to discover and remove the hex-encoded private key before decoding the remainder of the cipher text.

Figure 1. Test environment configuration

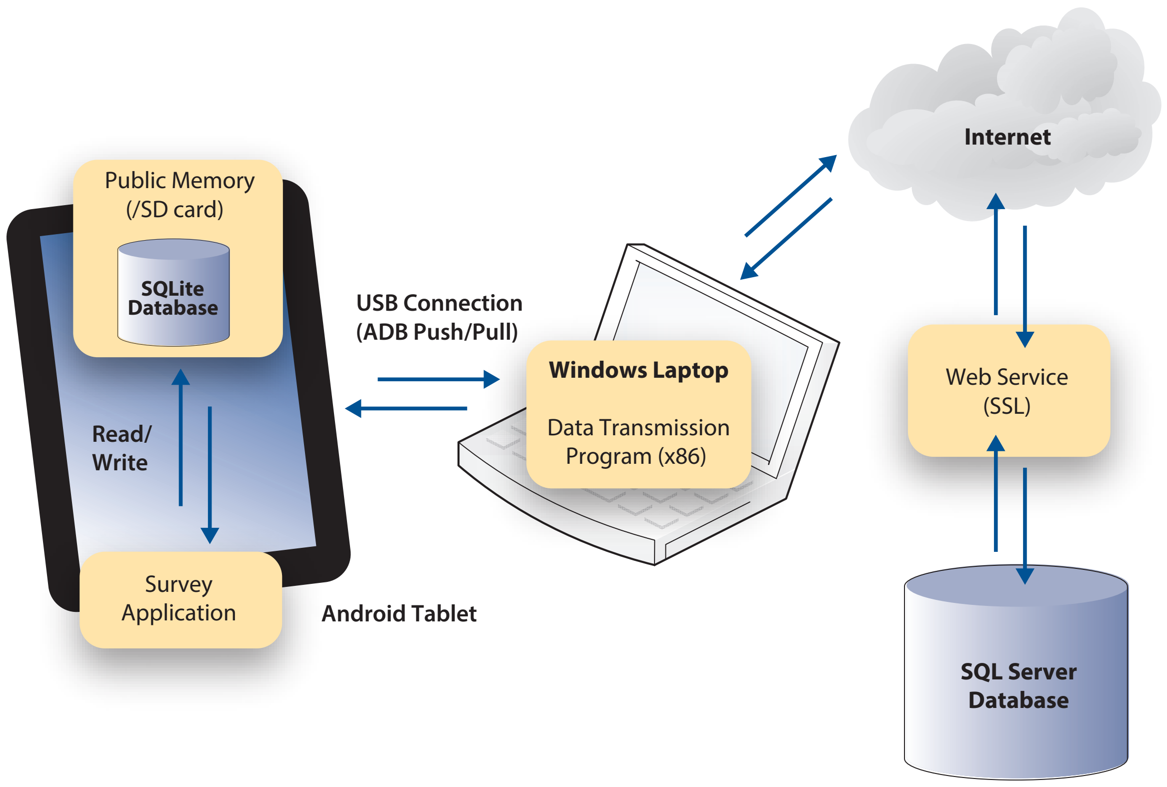
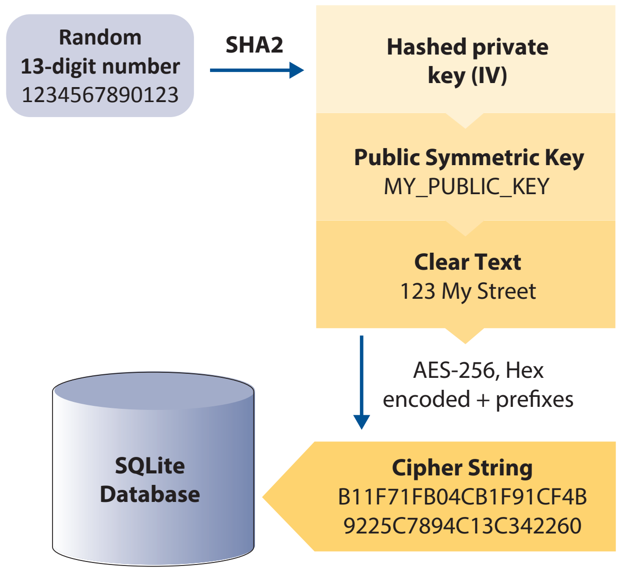


Figure 2. Encryption process



SHA = Secure Hash Algorithm 2;
AES-256 = 256-bit Advanced Encryption Standard.

A server-side process decrypts cipher text received during the transmission using the same public symmetric key value and private key (recovered from the cipher text) used during the encryption process. Note that the padding type and encoding of the clear string value must be matched for the data to be decrypted properly by the transmission program. Additionally, string attribute sizes of the client SQLite database and SQL server database need to be large enough to store the encrypted text. In the next section, we discuss how well the database attribute encryption worked and its limitations.

Results

To illustrate the result in a real-life app development, we used an example in which each household is a study case, and interviewers use Android devices running version 3.2 (Honeycomb) of the OS to display the household addresses to conduct interviews. The results demonstrated that the method works well.

Example

Initially, all study cases were stored on RTI's central database server. The street information for each household was encrypted with the shared symmetric key.

When an interviewer downloads a list of addresses from the server, the household data are transmitted and stored in the publically readable and writeable SQLite database on the handheld device, as shown in Figure 3.

Note that the content in the Street attribute is encrypted. By transmitting and storing data in such a manner, the household information is protected even if the interviewer loses the handheld device.

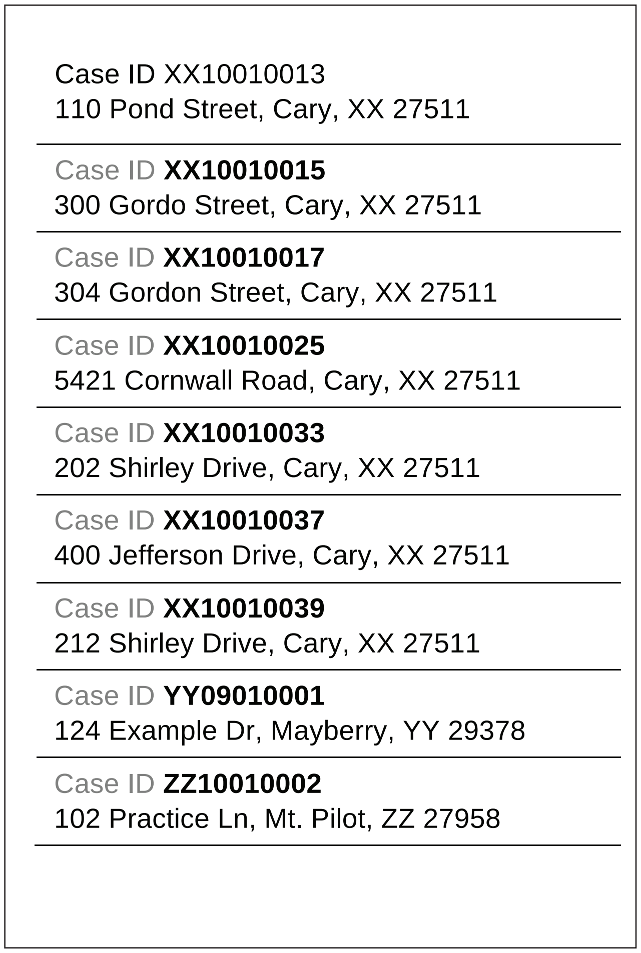
Before access to the app is granted, the interviewer must provide the symmetric key phrase used by the server program to encrypt data. The app uses this key phrase to retrieve the data from the SQLite database, decrypting the street data and displaying the list of households in the app, as shown in Figure 4.

Figure 3. Encrypted data sample

Case ID	Street #	Street	City	State	Zip
XX10010013	110	616D918DF787020CC836205C7975F9E9	Cary	XX	27511
XX10010015	300	AF6D27C81B91B53782107AA36D681D22	Cary	XX	27511
XX10010017	304	AF6D27C81B91B53782107AA36D681D22	Cary	XX	27511
XX10010025	5421	F22470396D17E34837394A76755C0BCC	Cary	XX	27511
XX10010033	202	52E751630AD19388E1A61838642F4FD5	Cary	XX	27511
XX10010037	400	687D99DA7D56591627750BE899E3E299	Cary	XX	27511
XX10010039	212	52E751630AD19388E1A61838642F4FD5	Cary	XX	27511
YY09010001	124	667811CB2C372B4C1511397FB634F06D	Mayberry	YY	29378
ZZ10010002	102	F5F395RC7B02CA90D75351343F80780F	Mt. Pilot	ZZ	27958

Prior to the field test, in-house security testing was conducted attempting to decrypt the data. All attempts failed without properly providing the correct key phrase. After the field test, all data were properly encrypted/decrypted with no major problems reported by the field staff. No missing or unrecoverable record data were logged as data anomalies by our data processing programs. Our experience with this app illustrates that the attribute-level encryption and decryption method effectively secures a subset of data in a SQLite database while keeping out-of-pocket expenses low.

Figure 4. Decrypted street data



Case ID XX10010013
110 Pond Street, Cary, XX 27511

Case ID **XX10010015**
300 Gordo Street, Cary, XX 27511

Case ID **XX10010017**
304 Gordon Street, Cary, XX 27511

Case ID **XX10010025**
5421 Cornwall Road, Cary, XX 27511

Case ID **XX10010033**
202 Shirley Drive, Cary, XX 27511

Case ID **XX10010037**
400 Jefferson Drive, Cary, XX 27511

Case ID **XX10010039**
212 Shirley Drive, Cary, XX 27511

Case ID **YY09010001**
124 Example Dr, Mayberry, YY 29378

Case ID **ZZ10010002**
102 Practice Ln, Mt. Pilot, ZZ 27958

Potential Problems

Although this method is cost-effective and easy to implement, we note its potential disadvantages, including performance/scalability, inability to sort raw data, unencrypted database schema, and key knowledge.

First, there is a performance penalty. Each time the app needs to display the addresses, rather than displaying the raw content from the database, the app must decrypt the street information for each household. Our testing showed that each decryption operation took ~22 milliseconds to complete (whereas encryption operations took ~15 milliseconds to complete). When displaying data for only one attribute needing decryption, as in our testing, the decryption latency was negligible. If multiple attribute values (scalability) need to be decrypted for display, then the decryption latency could become noticeable to the user.

Second, to store data in an encrypted fashion, we lose some ability to manipulate the data with SQL's built-in functionality. For example, in Figure 5, the app displays the household list in the order of Case ID. If we want to see street names that contain the string "Jones" or display the same list in the order of street name, which is the encrypted attribute value, we cannot use the SQL SELECT or the SQL ORDER BY clause when retrieving data from the database because the encrypted information will not meet the search criteria and may not keep the same order as the original data. Selecting or sorting encrypted data requires the implementation of homomorphic encryption algorithms,[14] such as CryptDB,[15] that are not available in SQLite or on the Android platform. Therefore, we would need to develop a scheme that uses temporary data decryption and additional attribute value indexing.

Third, the database schema is not encrypted. Nefarious users could mine information from the unencrypted attribute names to infer encrypted values.

Fourth, end users need to retain the knowledge of the key that is used to encrypt the data. They need to provide it to the Android app for the program to properly decrypt the content in the database.

End users need to protect knowledge of the key to maintain the security of the database.

While scalability limitations, the inability to select or sort raw data, the unencrypted database schema scenario, and key knowledge are the main drawbacks for our approach of attribute-level encryption, the simplicity of the design and implementation still made this method the best option for our study data.

Field Test Observations

To further test our implementation, we executed a field test. For the test, field interviewers used Android devices running version 3.2 (Honeycomb) of the OS to display the household addresses to conduct interviews. Interview test cases were processed much like a census: interviewers collected a roster of the household members, including age, gender, race, and military service. At the conclusion of the interview, the interviewer collected information (name, phone number) to allow for telephone verification of the interview and interviewer quality control. During the interview process, interviewers were allowed to correct and/or add missed addresses.

Our interview app employed the encryption/decryption process described in the Results section to ensure that all sensitive data (e.g., address, name, age) were secured. Address data was encrypted before being transmitted to the field, and the app encrypted all address changes and personal information collected before writing them to the SQLite database. At the RTI server end, a procedure ran hourly to detect encrypted data sent from the Android devices and to decrypt the data. Table 1 outlines the volume of data secured using our attribute-level encryption mechanism. Results of our test were positive with no reports of issues related to the encryption/decryption of sensitive data. Further, as expected, the app's user interface remained responsive; users did not report degradation.

The success of the field test proved that the attribute-level encryption method worked well and justifies implementation in a full-scale study. Because the algorithm is generic, it can be easily ported to other similar data collection projects.

Table 1. Volume of encrypted data

	Count
Unique addresses sent to field (client decryption)	9,651
Addresses modified (client encryption)	693
Addresses added (client encryption)	106
Roster names collected (client encryption)	411
Verification names collected (client encryption)	3,423
Phone numbers collected (client encryption)	3,267

Related Work

With regard to sorting encrypted data in an efficient manner, theoretical research is being conducted on homomorphic encryption.[16] With homomorphic encryption, the encrypted data are computed, and the result set is determined from the decrypted data, as if the data had not been initially ciphered. Although the result set performance could be considered slow,[14] the data could still be encrypted from unauthorized access, and the SQL SELECT and ORDER BY clauses could use data analysis. Further research to improve the performance of the fully homomorphic encryption mechanism will incorporate a CryptDB[15] framework. To increase the search speed efficiency of the encrypted data, specific encryption algorithms could be matched for a particular search operation. For example, an order-preserving encryption practice[17] could be matched to the SQL ORDER BY clause. Thus, encrypted data could be quickly and efficiently sorted without decryption. Most of the related work mentioned would need further investigation to implement in the SQLite database environment.

Future Work

We understand that attribute-level encryption of globally readable and writable databases on Android has some limitations. Some possible considerations for future work include Android intent communication, database file-level encryption, and verifying compliance with the Cryptographic Module Validation Program (CMVP). This list is a starting point to build on for improving security for sensitive data.

Move Database to Private Storage

To provide an additional layer of security, the database could be stored in the app's private, protected /data folder. To access data in this scheme, other apps and the Windows (x86)-based transmission program would be required to send a message, or intent, to our app requesting access to the data. Android intents are defined as hooks to pass specific definitions for intercommunication between apps and external programs. Data-layer intents could be used to request and pass information between our app, which is controlling the protected SQLite database, and the x86 transmission program. After receiving a data request intent, our app would return the requested data as a stream, as a temporary protected file, or by some other secure mechanism. Moving the database to private storage would require the x86 transmission program interface be retooled to handle the app's data delivery mechanism.

Database File-Level Encryption

Additional file-level encryption could be provided to prevent database schema manipulation. Because the whole database file would be encrypted, the database schema would be difficult to obtain for analysis. Protecting sensitive data with the database file encryption method plus using the attribute-level encryption would further enhance the security of the database.

Cryptographic Module Validation Program

When dealing with sensitive data, researchers should verify that their data collection device manufacturer and Android OS's cryptographic modules are tested and validated under the National Institute of Standards and Technology CMVP.[18] It is important to know the validation status to address project-specific data security requirements. If the validation is listed as revoked, then additional security and encryption algorithms would be needed.

References

1. Yarow J, Terbush J. Android is totally blowing away the competition [Internet]. Business Insider; 2011 Nov 15. Available from: http://articles.businessinsider.com/2011-11-15/tech/30400572_1_smartphone-android-ios
2. Wikipedia.com: Sandbox (computer security) [Internet]. [cited 2012 Aug 1]. Available from: http://en.wikipedia.org/wiki/Sandbox_(computer_security)
3. Enck W, McDaniel P. Understanding Android's security framework (tutorial). Presented at CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security; 2008 October 27-31; Alexandria, VA. [Online]. Available from: http://siis.cse.psu.edu/android_sec_tutorial.html
4. Pocatilu P. Android applications security [Internet]. Informatica Economica; 2011 [cited 2013 March 11]. Available from: http://revistaie.ase.ro/content/59/14%20-%20Pocatilu.pdf
5. Cannon T. Into the Droid—gaining access to Android user data [Internet]. 2012 DEF CON Conference, Las Vegas, NV: 2012 July 28. Available from: https://viaForensics.com/mobile-security-category/droid-gaining-access-android-user-data.html
6. SQLite.org. SQLite software [Internet]. 2012 Feb. Available from: http://www.sqlite.org
7. Developer.android.com. Storage options: using databases [Internet]. [cited 2012 Aug 8]. Available from: http://developer.android.com/guide/topics/data/data-storage.html#db
8. Hwaci.com. The SQLite Encryption Extension (SEE) [Internet]. 2012 Feb 1. Available from: http://www.hwaci.com/sw/sqlite/see.html
9. SQLCipher.net. SQLCipher: full database encryption for SQLite [Internet]. [cited 2012 Feb 1]. Available from http://sqlcipher.net/ or http://sites.fastspring.com/zetetic/product/sqlcipher

10. SQLCipher.net. Commercial support [Internet]. [cited 2013 Aug 27]. Available from http://sqlcipher.net/support/

11. Hwaci.com. SQLite Consortium [Internet]. [cited 2013 Aug 27]. Available from http://www.hwaci.com/sw/sqlite/member.html

12. Source.android.com. Notes on the implementation of encryption in Android 3.0 [Internet]. [cited 2012 Aug 1]. Available from: http://source.android.com/devices/tech/encryption/android_crypto_implementation.html

13. ITL.NIST.gov. Gaithersburg (MD): Security requirements for cryptographic modules [Internet]. 1994 Jan 11 [cited 2013 March 8]. Available from the NIST Information Technology Laboratory website: http://www.itl.nist.gov/fipspubs/fip140-1.htm

14. Cooney M. IBM touts encryption innovation; new technology performs calculations on encrypted data without decrypting it [Internet]. Computer World; 2009 June 25. Available from http://www.computerworld.com/s/article/9134823/IBM_touts_encryption_innovation

15. Popa RA, Redfield CMS, Zeldovich N, Balakrishnan H. CryptDB: protecting confidentiality with encrypted query processing. SOSP '11, Proceedings of the 23rd ACM Symposium on Operating Systems Principles; 2011 Oct 23-26; Cascais, Portugal. New York: ACM; p. 85-100.

16. Gentry C. Fully homomorphic encryption using ideal lattices. Proceedings of the 41st Annual ACM Symposium on Theory of Computing; 2009 May 31–June 2; Bethesda, MD. New York: ACM; p. 169-79. Available from http://dl.acm.org/citation.cfm?id=1536440

17. Boldyreva A, Chenette N, Lee Y, O'Neill A. Order preserving symmetric encryption. In Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT); 2009 April 26-30; Cologne, Germany. New York: Springer; p. 224.

18. CSRC.NIST.gov. Gaithersburg, MD: National Institute of Standards and Technology Computer Security Resource Center. Module validation lists [Internet]. [cited 2012 Aug]. Available from: http://csrc.nist.gov/groups/STM/cmvp/validation.html

Acknowledgments

Our thanks to the reviewing and editing provided by Martin Meyer, Craig Hollingsworth, Nanthini Ganapathi, Charlotte Scheper, and Patricia Smith.

www.rti.org/rtipress **RTI Press publication OP-0016-1309**