Protecting Sensitive Data in Android SQLite Databases Using TrustZone

Francis Akowuah, Amit Ahlawat and Wenliang Du
Electrical Engineering and Computer Science Department
Syracuse University
Syracuse, NY, USA
{feakowua, aahlawat, wedu}@syr.edu

***Abstract*—Applications use SQLite databases for storing data such as fitness/health information, contacts, text messages, calendar among others. Such information is sensitive and worth protecting. SQLite engine do not have built-in security to protect databases, rather, it relies on its environment such as the operating system to provide security for database content. While Android provides security mechanisms for SQLite databases, it has been shown to be inadequate. Proposed solutions are not able to protect sensitive database content when the OS is compromised. Also, some existing solutions fall short in protecting sensitive data if the SQLite database file is relocated to an environment that do not have any security restrictions. We propose a hardware isolation solution, leveraging ARM's TrustZone to protect sensitive content of databases. We design and implement a prototype system on Hikey development board to demonstrate that TrustZone can be integrated with Android to protect SQLite data. Evaluation results shows our system is practical in and do not break the design patterns in Android application development.**

***Index Terms*—TrustZone, SQLite, Database, Android**

I. Introduction

SQLite is a free in-process library that implements SQL database engine. It has been widely deployed in many systems to provide database functionality. One reason for SQLite's wide-spread use is due to its high storage efficiency, small memory needs and fast query operations. Android and iOS ship with SQLite and also make available a rich set of APIs for mobile application developers to implement databases in their apps. Common mobile apps that have natural use of databases include email, text messaging, settings, calendar among others.

As mobile device use have become a part of the modern life, some mobile applications end up storing private and sensitive information such as blood pressure and sugar, passwords, SSN among others in SQLite databases. SQLite databases are stored as a cross-platform file and data is stored without obscurity. Hence, anyone or application that has direct access to the database file can read and modify the whole database content. This calls for measures to protect sensitive data stored in databases.

Morever, the SQLite engine, unlike client-server DBMS such as Oracle and SQL Server, do not not have built-in security such as authentication, authorization or crypto systems, therefore, it relies on its environment (such as the operating system) to provide the required security. The Android operating system meets this need by providing multiple security mechanisms for SQLite databases and the application as a whole. First, Android provides a User-ID (UID) based access control policy on the app's installation directory. That is, each application is sandboxed and therefore it is only the application process that can access the content of its installation directory. When an application uses SQLite, the SQLite database, which is a cross-platform file is stored in the application's installation directory. Therefore SQLite databases are isolated from other apps on the Android mobile device. Secondly, Android uses its permission framework together with Content Providers to provide access control for SQLite databases. When an application wants to share its database with other application, it uses Content Providers to do so. The app developer may use Android-defined or user-defined permissions to restrict access to the SQLite database. For example, READ_CONTACTS permission only allows the contacts database to be read; it cannot be written to. However, these Android security mechanisms are not sufficient in the event where a malicious application compromises the operating system to gain root privileges. It can bypass all the sandboxing security features and gain direct access to the database file. Davi et al [1] have shown that Android's sandbox model is inadequate since applications can escalate the permissions that they have been granted. [2] also show rooting attacks are possible on Android.

In order to enhance the security of SQLite database, encyption and access control solutions have been proposed. Encryption can be applied to a database' field or record as well as the databse file as seen in [3]–[9]. Although encryption is a viable solution to protect database content, crypto key management has been a difficult task for many developers since the key is usually stored in the executable file or generated in the same process or (compromised) computing environment. This has enabled attackers to discover keys as seen in [10].The malicious app can gain access to the encryption key and then decrypt the database. Our solution improves upon pure encryption solutions by isolating the environment in which the crypto key is stored and used. Android also provides File Disk Encryption (FDE) which encrypts all the data on the filesystem. However, FDE protects data only when the mobile device is locked; FDE is disabled when the device owner unlocks the device and all data becomes plaintext.

Other than crypto solutions, the research community has proposed a couple of access control solutions. Mutti et al

ISBN: 1-60132-488-X, CSREA Press ©

[11] integrated SQLite and SELinux in order to enforce fine grain access control at the lowest level in the database based on context security. Their solution requires changes to both the OS and the SQLite library. Similarly, Ali-Gombe et al [12] proposed a solution that defines access control for both database schema and entities by merging static bytecode weaving and database query rewriting to achieve low-level access control for Android native providers at the application level. Hence their solution do not make any change to the underlying OS, rather, it provides a Controller stub that statically weaves into the target application. It also provides Controller interface that is used for setting access levels. Unfortunately, these proposed access control systems are weak against the adversary that has gained elevated privileges. Also, they fail to protect sensitive data if the database file is moved to a different environment that do not have proposed access controls defined. For instance, the adversary can send the database file to a remote computer. For stronger privacy and security guarantees in a compromised OS and protecting the data whenever it is moved to a new environment, we believe that isolation solution is better than access controls mechanisms.

Therefore, we propose column-level encryption backed by a hardware isolation solution that protects sensitive database content in a compromised environment. We undertake the non-trivial challenge to integrate TrustZone with Android such that we can maintain the app's interaction with SQLite databases across two operating systems. We securely enter the sensitive data in the secure world and then ensure that the crypto key and sensitive data stays in the TEE always.

Our contributions include the exploration of integrating ARM's TrustZone with Android to protect sensitive data on the Android OS. Secondly, we design and implement a prototype system on Hikey board running Android 7.1 and OPTEE OS. Lastly, we provide security analysis and evaluation of our system.

Paper is outlined as follows. §II gives background on SQLite and ARM TrustZone while §III discuss our threat model and assumptions. §IV and §V provides the design principles and system overview respectively. In §VI, we give the security analysis of the system. Evaluation, related work and conclusion are discussed in §VII, §VIII and §IX.

II. Background

We provide some background on SQLite databases and the storage options available in Android for its application. Also, we briefly describe the ARM TrustZone technology.

A. Storage Options For Android Apps

The Android OS provide multiple ways to store persistent data. The options include Shared Preferences, Internal Storage, External Storage, SQlite Databases and Network Connection (on web server). External Storage make data publicly available on the device whereas Shared Preferences, SQLite Databases and Internal Storage keep the data private to the application. This work only focuses on SQLite Databases.

B. SQLite

SQLite [13], the most widely deployed software in the world, is a relational database management system (RDBMS) similar to MySQL, PostgreSQL and SQL Server. However, SQLite does not have the client-server model that the aforementioned RDBMSs have. Hence, SQLite do not require a server, administration tasks or any complex configuration process. Yet, applications that embed SQLite still enjoy all the power of a relational database. On the Android platform, SQLite is used to manage system and user databases storing several types of information including contacts, SMS messages, and web browser bookmarks. Besides its use on mobile devices, SQLite has been used in aviations, electronics, health applications etc. The entire database is stored in a single cross-platform file and runs in the same process as the app. SQLite is free for both commercial or private purposes.

SQLite is an ACID compliant database that implements most of the SQL standard. It does not implement SQL commands such as RIGHT OUTER JOIN, FULL OUTER JOIN, GRANT and REVOKE. SQLite VIEWs are read-only. The interested reader is referred to [13] for a complete list of SQL features that are not implemented in SQLite.

Android and iOS provide strong support for applications to use SQLite by making available application programming interfaces (APIs) for creating, querying and modifying databases. Android provides many classes in *android.database.sqlite* package for database management. IOS also provides a similar support through the *CoreData* framework.

SQLite is not the only data persistence solution on the Android OS even though this research focuses on SQLite. Alternatives include Realm DB [14], Couchbase Lite [15], Berkeley DB [16], ORMLite [17] among others. Android has introduced the Room Persistence Library [18] to allow fluent database access. *Room* is an abstraction layer over SQLite. While the Android app developer has the option to the SQLite APIs and Room APIs, the documentation recommends the latter.

C. ARM TrustZone

TrustZone (Fig. 1) is ARM's trusted execution environment (TEE) technology that enables CPU to run in two modes: secure and normal modes. Normal world or mode is where the normal computations of the CPU takes place. System and third-party applications run in the normal world. Secure world, on the other hand, is used for secure and private-sensitive computations. Similar to the normal world, the secure world has user space and kernel layer. Applications that run in the secure world's user layer is called Trusted Applications (TA) or Trustlets. Many CPUs shipped with mobile devices have Trustzone. Currently, only vendors and OEMs utilize the security guarantees of the technology since it is locked when the device ships.

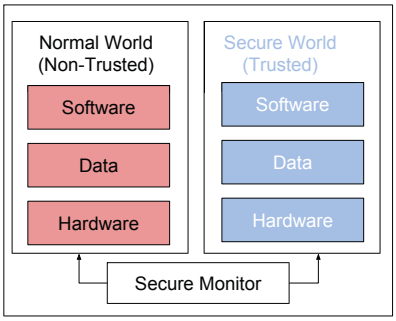

Fig. 1. ARM TrustZone

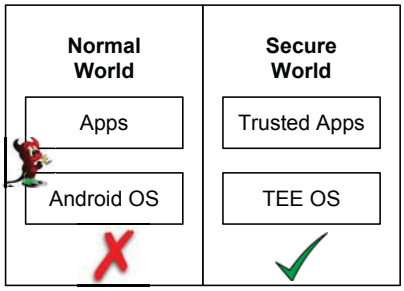

Fig. 2. Threat model

III. Threat Model And Assumptions

A. Attacker Model

We consider a threat model (depicted in Fig. 2) whereby the normal world is compromised. Although the attacker can attack the upper layers of Android enabling privilege escalation by leveraging attacking techniques in [1], [19], [20] and [21], he does not gain root privileges. We consider a strong adversary who is also able to attack the lower layers of android enabling him to gain root privileges using techniques as those discussed in [22] and [23]. Thus, the attacker is able to bypass all the security mechanisms in the normal world operating system (Android) and for that reason we do not trust the normal world's user-space applications and operating system. We, however, trust the secure world applications and operating system. The TrustZone technology provides hardware isolation that do not allow our strong adversary to access resources in the TEE.

Lastly, the attacker is able to decompile (using tools such as [24] and analyze the Android application's APK file allowing attacker to gather information about database operations in the application. The attacker is able to learn about any secrets the developer mistakenly placed in the source code such as a crypto key or password.

The attacker's intent is to read the sensitive content of the SQlite database and/or to relocate the database file to an environment that has no restrictions on reading the database content. The attacker is also interested in carrying out static and dynamic leakage attacks [6] by observing the ciphertext values to learn a pattern or infer plaintext values.

B. Assumptions

Our system includes Trusted Applications (TA) that run in the secure world. The TAs do not contain a large amount of code. We assume that there are no vulnerabilities in our TAs.

Sensitive data are inputted and encryted in the secure world of our system. The data should not be given to the normal world for display purposes. Therefore, we assume SchrodinText [25] is implemented on the Android OS so that sensitive data can be securely be displayed to the user without giving it the normal world.

IV. Integrating TrustZone with Android

The Android Framework support applications to use SQLite by providing APIs in the *android.database.sqlite* package. The APIs interact with the SQLite library which is found in the Native Library layer via Java Native Interface (JNI). SQLite library returns the results of the database operations to the framework via JNI. Finally, the result is given to the application. Fig 3a demonstrates how android apps interact with the SQLite library.

As seen above, no trusted execution environment (TEE)is involved in the app's interaction with SQLite. We have the non-trivial challenge to integrate TrustZone with Android such that we can maintain the app's interaction with databases across two operating systems (Android and Secure OS) as shown in Fig 3b. We want this integration to be transparent to the app developer as much as possible such that he does not bother about how his apps interacts with TrustZone.

To achieve our goal, we consider the following in our design: (1) SQLite database placement, (2) securely inputting the sensitive data (3) building secure world components to interact with normal world app and (4) securely displaying the sensitive data. We discuss items (2) to (4) in subsequent sections.

The placement of the SQLite database can impact the system's performance. We have the option of placing the whole database in the secure world, splitting the database into sensitive and non-sensitive databases or the more challenging option of maintaining one database in the normal world whilst protecting the sensitive data from the secure world. We choose to design and implement the last option. Our design aims to have (1) a small Trusted Computing Base (TCB), (2) require minimal changes to the OS, (3) minimize the number of switches between the normal and secure worlds and (4) avoid query rewriting when possible.

V. System Overview

In this section, we describe our system components and how they interact. Fig. 4 shows the high-level view of our system.

A. System Components

We discuss the various components that make up our proposed system. We describe the Trusted Applications (TA) that we implemented. A TA is an application that runs in the user-space of the Secure OS. TAs have the full power of the

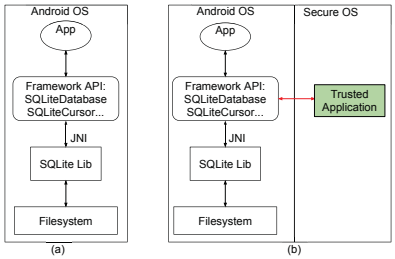

Fig. 3. (a) How applications interact with the SQLite Library. (b) Integration of Android and TrustZone

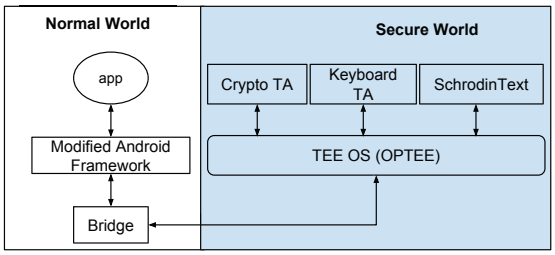

Fig. 4. System overview

CPU and the memory of the device. SchrodinText component is assumed to be already present on the experimental setup. Interested reader is referred to [25] for details on SchrodinText.

1) Modified Android Framework: We introduce new API to the *android.database.sqlite.SQLiteDatabase* class. Methods *insertTZ()*, *queryTZ()* and *updateTZ()* are used for inserting, querying and updating sensitive fields respectively. Considering the intent of the attacker in our attacker model, we do not make changes to the delete functionality. Applications that do not have any sensitive field to protect can still use stock Android's APIs to carry out SQLite operations.

Given that we consider a Normal World OS that is compromised requires that we protect the sensitive data that the user enters using the keyboard. Leveraging techniques described in [26], we moved the keyboard UI to the Secure World while preserving the binding between the keyboard UI and Android's UI widget called *EditText*. In Android, EditText triggers the keyboard UI. To trigger the secure keyboard UI, we extend EditText to include a type called *secure*. The developer uses the extended EditText (with *secure* type) to communicate to the Android Framework that he intends to capture a sensitive data that will be stored in the SQLite database. The app's secure keyboard request is sent via a modified *InputMethodManagerService* (IMMS) Framework service to a new proxy IME system app which then interacts with the Keyboad TA (§V-A3). The keyboard TA displays the secure keyboard UI. We add LED light to the Secure World which is only lit when the CPU is operating in the Secure mode indicating to the user that keyboard input will be captured in the secure world.

2) Crypto TA - Encrypting /decrypting sensitive data: : Crypto TA is a trusted application (written in C with 615 LOC) that runs in the secure world. As its name suggests, it is responsible for encrypting and decrypting sensitive data using AES 256 CBC mode. It generates random IV for each sensitive data to thwart static and dynamic leakage attacks discussed in [6]. Let C represent the sensitive data's ciphertext. The TA concatenates the random IV (IV) with the ciphertext. Let $S = C + IV$. Crypto TA encodes S before sending to the normal world ($E = Enc(S)$). The encoding step is done to prevent loss of data as it moves between worlds. Without encoding, we realized C could not be decrypted correctly after retrieving from the Normal World. The ciphertext contained non-printable characters that the normal world could not render correctly.

To decrypt the data, the Crypto TA extracts the IV and C from E after which it applies the decryption function.

3) Keyboard TA: The Keyboard TA communicates with the Keyboard UI to obtain the sensitive data from the user. After capturing the user input, the typed sensitive data is forwarded to the Crypto TA for encryption and encoding. Crypto TA returns E (§V-A2) to the Keyboard TA which is then returned to the proxy IME app. Using its binding with the app's Editext, the proxy IME app returns E to the Secure EditText. The app code can then access E by calling EditText's *getText()* method.

4) Bridge: The Bridge is a native component that invokes the secure world. It has 634 lines of C code. Apps interact with the Bridge via TEEBrideManager which serves as an interface for the TEEBridge service. TEEBridge is a system service that we added to the Android Framework.

B. System Workflow

Here we describe how our system works. Our system currently do not support SQL commands that includes a sensitive column in the WHERE clause. Allowing filtering on a sensitive field can enable the attacker to make a correct guess.

1) Sensitive data insertion:: User presses on secure EditText widget and obtains a secure keyboard UI from the secure world. This switches the CPU to the Secure Mode. User enters sensitive data and dismisses the secure keyboard. The Keyboard TA invokes the Crypto TA to encrypt and encode the inputted data as described in §V-A2. After receiving results from the Crypto TA, the Keyboard TA returns the encoded ciphertext (E) to the secure EditText. The app code obtains the Secure EditText's text by caling *getText()*. Using *android.database.sqlite.SQLiteDatabase.insert()* method, the app inserts the text obtained into the SQLite database. Fig 5 shows this workfow.

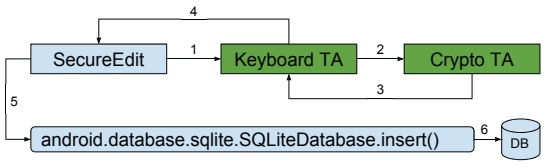

Fig. 5. Workflow for inserting sensitive data

2) Retrieving and displaying sensitive data:: Applications display database content to user in UI widgets such as TextBox. When the OS is compromised the attacker is able to

steal sensitive data when it is being displayed in an UI widget. For this reason, we cannot decrypt the sensitive data in secure world and send it to the normal world for display purpose. We need a solution that can display the data in the secure world without giving the plaintext to the normal world. SchrodinText [25] provides such solution and as stated in §III-B, we assume that SchrodinText is already implemented on Android.

We add queryTZ() method to the *android.database.sqlite.SQLiteDatase* class. This method retrieves the encoded ciphertext E from the database and calls SchrodinText's *setCiphertext* to bind E to the SchrodinTextView (modified version of Android's UI widget *TextView*). SchrodinText interacts with the CrytoTA to decrypt the sensitive data. SchrodinText then displays the plaintext from the Secure World without giving the data to the Normal World. This workflow is shown in Fig. 6

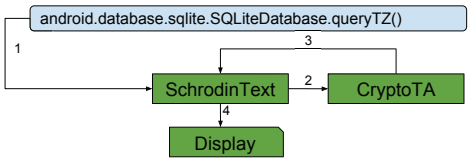

Fig. 6. Workflow for retrieving and displaying sensitive data

3) Updating sensitive data:: Our system allow sensitive data to be updated. This process is similar to inserting sensitive data (§5). In this workflow, we assume the row data to be updated is loaded in UI widgets and that the sensitive data is attached to secure EditText. This ensures that data is securely captured in the secure world. After data capture, app code calls *android.database.sqlite.SQLiteDatabas.update()* method to update.

C. Implementation

We implement our system on LeMaker's Hikey development board which is powered by the Kirin 620 SoC with octa core ARM Cortex-A53 64-bit CPU up to 1.2GHz and high performance Mali450-MP4 GPU. Hikey also has 1GB/2GB LPDDR3 DRAM (800MHz) and 8GB eMMC storage on board. We run Android 7.1 and Linaro's OPTEE [27] in the normal world and secure world respectively.

VI. Security Analysis

In this section, we present the security analysis of our system. Our analysis assumes that the TrustZone platform is trusted and the secure boot process has initialized the integrity-verified OPTEE OS. We do not consider hardware attacks, side channel attacks and DOS attacks.

A. Database Security Analysis

The intent of the adversary is to bypass all security mechanisms in the Android operating system in order to read the sensitive content of the SQLite database. Our goal is not to prevent the attacker from getting access to the data, rather, we want to prevent him from having knowledge of the sensitive data. In our system, the sensitive content was entered and

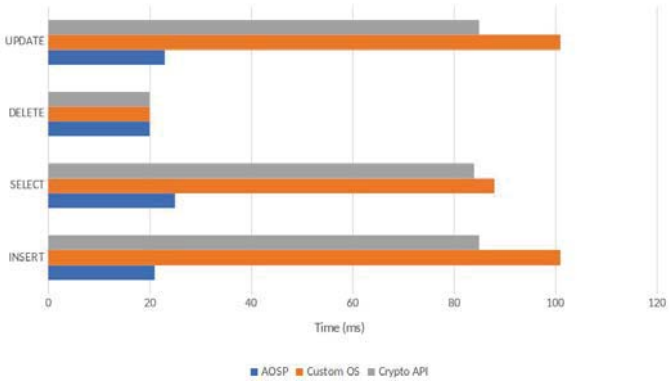

Fig. 7. Performance measurement

encrypted in the secure world where the adversary is unable to access. The crypto key stays in the secure world and it is never given to the normal world. Therefore, when the adversary gets direct access to the SQLite database file, he cannot decrypt and read the sensitive data since the key is hidden secure world.

The adversary is also interested in observing patterns in ciphertext to infer plaintext. We prevent this or make it harder by generating random IVs for each sensitive data so that two plaintext of the same value do not generate the same ciphertext.

Moreover, the adversary may send database file to a remote computer to evade other security restrictions existing in the android OS. Our system ensures the crypto key does not leave the secure world making it impossible for adversary to send both the crypto key and the database file together to the remote computer. Unlike most encryption methods where the key is generated and/or stored in the same execution environment (normal world) as the ciphertext, our hardware isolation guarantees that key is always isolated from the ciphertext.

B. Keyboard Security Analysis

When the keyboard UI is called, it switches the CPU to the Secure Mode where the normal world cannot monitor any of the secure I/O ports. This guarantee is given by the ARM TrustZone from the hardware. Thus, the normal world adversary cannot switch the screen input or screen USB input thereby preventing keylogging attacks. To prevent screen capture attacks, TrustZone also blocks the normal world from switching the HDMI output of the screen. Lastly, we do not disclose the input to the normal world.

VII. Evaluation

In this section, we evaluate our design with regards to ease of adoption, performance and effectiveness. Using real world open source applications, we tested how easy it is for developers to adopt our system to protect sensitive data in their database. Also, we measured the performance overhead that is introduced as a result of the changes we made to the Android framework and the integration of ARM TrustZone. Lastly, we evaluate the effectiveness of our system in a few use cases.

A. Performance Measurement

We evaluate performance in terms of the SQL language's CRUD commands, namely, INSERT, SELECT, UPDATE and DELETE. We compare the time it takes to execute the CRUD commands in three scenarios: (1) CRUD operations without any encryption, (2) CRUD operations using Android's Crypto API and (3) CRUD operations using our system (encryption with hardware isolation)

1) INSERT and UPDATE: We observe an overhead averaging 100 milliseconds when inserting or updating a record. This is as a result of the time that it takes to switch to and back from the secure world. Further, we observed the encryption algorithm in the TA takes about 18 ms. We used the encryption APIs implemented by OPTEE OS in our TA. overhead was however not noticeable on our testing app.

2) DELETE: Deleting a record from a table did not incur overhead because we maintain only one database in the normal world. So deletion occurs in the normal world without secure world involvement.

3) QUERY: The SELECT command itself did not introduce any performance overhead because it did not involve any transition to the secure world. However, obtaining the query results from the SQLiteCursor object incurs an overhead as the ciphertext is decrypted in the secure world.

Fig.7 gives a summary of the performance measurement. Chart clearly shows that the overhead our system incurs is due to the world switch since the time for using Android crypto APIs is not significantly different from the time that our system incurs.

B. Ease of Adoption

Here we evaluate how easily developers may adopt our system to protect their sensitive database content. Developers need to only call our system's API and this usually require a line or few lines of code. Code snippet below shows how a developer would use our secure EditText that would trigger the keyboard UI in the secure world.

```
 <EditText
    android:id="@+id/editText"
    android:inputType="secure"  ①
}
```

Line ① sets the inputType to *secure*. This is the only line needed to allow the user to capture the sensitive data in the secure world.

C. Impact on Content Providers, Loaders and Adapters

SQLite is the popular backend for Content Providers. Content Provider is an interface that allow applications to share their data in a controlled manner. We evaluate the system to see if our changes to the Android framework breaks that design pattern. Also we wanted to ensure that our system still works with Loaders. Loader classes such as CursorLoader and AsyncTaskLoaders are used to perform slow operations so that it does not block the UI.

We developed a simple application that perform SQL CRUD operations on a content provider application that uses our system. Further, we modified an open source application to use our test Content Provider's URI. The results show our system do not break the behavior of content provider in android app development. Developers do not have to make any change to the way they have been writing their content provider applications.

VIII. Related Work

SESQLite [11] incorporated SELinux, a mandatory access control system, with SQLite. Our work differs from SESQLite in two ways. First their goal was to provide a fine-grained access control on SQLite database, whereas our goal is to explore how Trustzone is used to preserve the confidentiality of SQLite database content. Secondly, SESQLite is an access control solution whereas we propose an isolation solution. Essentially, if the database file is copied to another device, SESQLite fails to protect the sensitive information since the access control policy file will be missing from the new environment. Our solution, on the other hand, protects sensitive content since the key that encrypts on the sensitive data stays in the secure world of the original device. Ali-Gombe et al [12] also propose an access control solution with the goal of providing a fine-grained access control for android's native content providers. [12] differs from [11] in terms of where the access control was implemented. SESQLite integrated the access policies in the database engine while [12] enforced the access constraints at the application level by hooking the CRUD method calls and forcing query-rewriting where necessary. Essentially, considering the Android stack, while [12]'s solution applies to the application level, our solution applies to the framework level and [11] applies their solution to the SQLite library which is found under Native Library layer. [28] propose an architecture for a secure database environment on Android by entirely separating the database management system from the application domain.

SQLite's source code is open-source and can be extended to achieve many purposes such as leaving hooks to implement encryption mechanisms [29]. These hooks have been implimented in [30] and [31] to provide encryption support for SQLite. Further, SQLite Encryption Extension (SSE) [32] also supports the encryption of the whole database content including the metadata. It allows SQLite to read and write encrypted database files. Unlike the regular publicly available SQLite, SEE's source code and precompiled binaries are only available to licensees. Whereas SEE can read and write files created by the public version SQLite, the vice-versa is not true. SQLCipher [9] is an open source and commercial extension to SQLite that provides transparent 256-bit encryption of database files. While our work implements encryption, it is done by a trusted application in the secure world of TrustZone thereby hardening the security of the key. We do not put extra burden on SQLite as we want it to maintain its acclaimed fast and lightweight features.

IX. Conclusion

In this paper, we have discussed how ARM TrustZone is integrated with Android to protect sensitive data in SQLite databases. In our prototype system, we implemented a column-level encryption backed with hardware isolation by making changes to the Android Framework code. Evaluation results show an overhead in performance but we conclude our system is practical as the overhead incurred was not significantly different from the overhead incurred when using Android's Crypto APIs. We posit that the overhead which stems from context switching between worlds can be reduced as hardware technology improves. We envisage our system to be deployed by mobile device vendors. Our solution can also enable app developers to build apps that are HIPAA compliant.

References

[1] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proceedings of the 13th International Conference on Information Security*, ser. ISC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 346–360. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949317.1949356

[2] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 95–109.

[3] D. E. Denning, *Field Encryption and Authentication*. Boston, MA: Springer US, 1984, pp. 231–247. [Online]. Available: https://doi.org/10.1007/978-1-4684-4730-9_19

[4] C.-C. Chang and C.-W. Chan, "A database record encryption scheme using the rsa public key cryptosystem and its master keys," in *2003 International Conference on Computer Networks and Mobile Computing, 2003. ICCNMC 2003.*, Oct 2003, pp. 345–348.

[5] Microsoft. (2018, 12) Encrypt a column of data. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/encrypt-a-column-of-data

[6] E. Shmueli, R. Vaisenberg, Y. Elovici, and C. Glezer, "Database encryption: An overview of contemporary challenges and design considerations," *SIGMOD Rec.*, vol. 38, no. 3, pp. 29–34, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1815933.1815940

[7] G. I. Davida, D. L. Wells, and J. B. Kam, "A database encryption system with subkeys," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 312–328, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/319566.319580

[8] E. Shmueli, R. Waisenberg, Y. Elovici, and E. Gudes, "Designing secure indexes for encrypted databases," in *Proceedings of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, ser. DBSec'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 54–68. [Online]. Available: http://dx.doi.org/10.1007/11535706_5

[9] SqlCipher. (2017, 12) Full database encryption for sqlite. [Online]. Available: https://www.zetetic.net/sqlcipher/

[10] S. Mendoza. (2018, 06) Samsung pay: Tokenized numbers, flaws and issues. [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Mendoza-Samsung-Pay-Tokenized-Numbers-Flaws-And-Issues-wp.pdf

[11] S. Mutti, E. Bacis, and S. Paraboschi, "Sesqlite: Security enhanced sqlite: Mandatory access control for android databases," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/2818000.2818041

[12] A. Ali-Gombe, G. G. Richard, III, I. Ahmed, and V. Roussev, "Don't touch that column: Portable, fine-grained access control for android's native content providers," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '16. New York, NY, USA: ACM, 2016, pp. 79–90. [Online]. Available: http://doi.acm.org/10.1145/2939918.2939927

[13] SQLite. (2017, 12) About sqlite. [Online]. Available: https://www.sqlite.org/about.html

[14] Realm. (2018, 02) Realm database. [Online]. Available: https://realm.io/products/realm-database/

[15] Couchbase. (2018, 02) Getting started - android. [Online]. Available: https://developer.couchbase.com/documentation/mobile/current/couchbase-lite/java.html#getting-started

[16] O. Berkeley. (2018, 02) Building berkeley db for android. [Online]. Available: https://docs.oracle.com/cd/E17276_01/html/installation/build_android_intro.html

[17] ORMLite. (2018, 02) Ormlite - lightweight java orm supports android and sqlite. [Online]. Available: http://ormlite.com/sqlite_java_android_orm.shtml

[18] A. Doc. (2018, 02) Save data in a local database using room. [Online]. Available: https://developer.android.com/training/data-storage/room/

[19] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028089

[20] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[21] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android." in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2012.

[22] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 95–109.

[23] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046618

[24] Apktool. (2018, 06) A tool for reverse engineering android apk files. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[25] A. Amiri Sani, "Schrodintext: Strong protection of sensitive textual content of mobile applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17. ACM, 2017, pp. 197–210. [Online]. Available: http://doi.acm.org/10.1145/3081333.3081346

[26] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system." in *Proceedings of The 16th ACM International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18, 2018.

[27] S. Mendoza. (2018, 06) Open portable trusted execution environment. [Online]. Available: https://www.op-tee.org/

[28] J. H. Park, S. M. Yoo, I. S. Kim, and D. H. Lee, "Security architecture for a secure database on android," *IEEE Access*, vol. 6, pp. 11 482–11 501, 2018.

[29] H. Liu and Y. Gong, "Analysis and design on security of sqlite," 07 2013.

[30] Z. Yuehua and Z. Weiling, "Design and realization of database encrypt module based on sqlite," *Computer Engineering and Design*, vol. 29, no. 16, pp. 4132–4134, Aug 2008.

[31] L. Shunhe and L. Jiajin, "Analysis and research of the encryption method for sqlite," *Computer Applications and Software*, vol. 25, no. 10, pp. 70–72, Oct 2008.

[32] SEE. (2017, 12) Sqlite encryption extension software configuration management system. [Online]. Available: https://www.sqlite.org/see/doc/trunk/www/index.wiki