SECURITY ARCHITECTURE FOR A SECURE DATABASE ON ANDROID

Seminar Report

Submitted in partial fulfillment of the requirements for the award of degree of

BACHELOR OF TECHNOLOGY

In

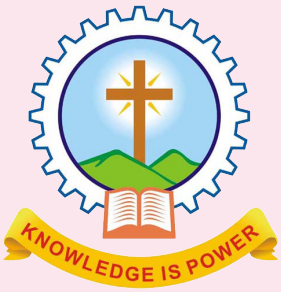
COMPUTER SCIENCE AND ENGINEERING

of

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

Submitted By

JOSEPH ASHWIN KOTTAPURATH



Department of Computer Science & Engineering
Mar Athanasius College Of Engineering Kothamangalam

SECURITY ARCHITECTURE FOR A SECURE DATABASE ON ANDROID

Seminar Report

Submitted in partial fulfillment of the requirements for the award of degree of

BACHELOR OF TECHNOLOGY

In

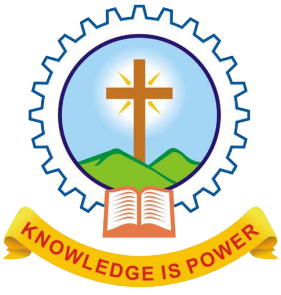
COMPUTER SCIENCE AND ENGINEERING

of

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

Submitted By

JOSEPH ASHWIN KOTTAPURATH



Department of Computer Science & Engineering
Mar Athanasius College Of Engineering Kothamangalam

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
MAR ATHANASIUS COLLEGE OF ENGINEERING
KOTHAMANGALAM

CERTIFICATE

This is to certify that the report entitled **Security Architecture for a Secure Database on Android** *submitted by* **Mr. JOSEPH ASHWIN KOTTAPURATH**, *Reg.No.***MAC15CS034** *towards partial fulfillment of the requirement for the award of Degree of Bachelor of Technology in Computer science and Engineering from APJ Abdul Kalam Technological University for December 2018 is a bonafide record of the seminar carried out by him under our supervision and guidance.*

.....	.....	.....
Prof. Joby George	**Prof. Neethu Subash**	**Dr. Surekha Mariam Varghese**
Faculty Guide	***Faculty Guide***	***Head of the Department***

Date:

Dept. Seal

ACKNOWLEDGEMENT

First and foremost, I sincerely thank the 'God Almighty' for his grace for the successful and timely completion of the seminar.

I express my sincere gratitude and thanks to Dr. Solly George, Principal and Dr. Surekha Mariam Varghese, Head Of the Department for providing the necessary facilities and their encouragement and support.

I owe special thanks to the staff-in-charge Prof. Joby george, Prof. Neethu Subash and Prof. Joby Anu Mathew for their corrections, suggestions and sincere efforts to co-ordinate the seminar under a tight schedule.

I express my sincere thanks to staff members in the Department of Computer Science and Engineering who have taken sincere efforts in helping me to conduct this seminar.

Finally, I would like to acknowledge the heartfelt efforts, comments, criticisms, co-operation and tremendous support given to me by my dear friends during the preparation of the seminar and also during the presentation without whose support this work would have been all the more difficult to accomplish.

ABSTRACT

Most mobile Operating Systems stores personal data in databases and provide APIs for applications to access it. Android stores data as plaintext in its database. As a result, the database content can be leaked unintentionally through several vulnerabilities. This paper proposes a security architecture to construct a secure database environment on Android. To this end, the database system is entirely separated from the application domain. This is the first such design for localized mobile databases. The separated database system manages a database with encryption. By delivering the responsibility over to the system, this separation enables application developers to be free from the difficult task of managing the security of the database. The proposed system also provides tight access control over a database by using a runtime information of an application. Thus, access is granted to a database if the application has the correct uid, regardless of the identity of the application. The proposed method creates a one-to-one pairing between the application and its database, and ensures that database access is granted only to the owner application. To evaluate the feasibility of the proposed architecture, a series of experiments on the prototype implementation is conducted. The results show that the proposed secure database architecture is feasible with acceptable overhead.

Contents

List of Figures

List of Abbreviation

TEE	Trusted Execution Environment
REE	Rich Execution Environment
FDE	Full Disk Encryption
UID	Unique Identifies
APK	Android Package Kit
SDBD	SecureDatabase System Daemon
API	Application Programming Interface
OS	Operating System
MAC	Media Access Control
FLASK	Flux Advanced Security Kernel

Introduction

In recent years, mobile devices have largely replaced personal computers as the reference platform to carry out daily activities. With the increase in usage of mobile devices, the amount of personal/business data stored in mobile devices has increased accordingly. These data are commonly stored as files or databases. Thus, a mobile OS provides several data storage APIs to store data. In particular, a database can efficiently manage an organized data set; thus, most Android applications (we use application/app interchangeably in this document), including default applications such as Contacts, Calendar, and SMS, utilize databases to store data. Because personal/business data on mobile devices may include confidential information, major mobile Operating Systems offer various encryption functionalities such as per-file encryption and full disk encryption. Google introduced Full Disk Encryption (FDE) from Android 3.0 onwards. FDE provides on-the-fly encryption and decryption that operates under the filesystem for all read/write operations on the device.

However, FDE protects the data only when the data are encrypted. When a user unlocks an Android device, all data in the device are decrypted. That is, FDE protects the data against off-line attacks on a locked device but not against attacks during operation. On an unlocked device, anyone with the proper permissions can access and manipulate all data in the database even if FDE is applied.

As mentioned above, FDE alone can not protect the data from exposure. To protect the data stored in the database while developing an Android application (user app or system app), developers must implement a self-solution for database encryption or use a third-party library. Because app developers work mostly in the application domain, they face some limitations that cannot be easily overcome. One such limitations is key management.

Most apps that include encryption solutions require a user to enter a password to generate a secret key to be used for encryption/decryption. Otherwise, hard-coded secret information for key generation is used to avoid the inconvenience of entering a password every time. If an app requires a password from a user, the user must enter a password each time an app is launched, which is a great inconvenience. If an app uses hard-coded secret information or device-specific

information (e.g. DeviceID, MAC address) for key generation, there is a risk that such information will be exposed by reverse engineering. In addition, this approach can have a significant ripple effect, because the exposed confidential information might be equally applied to all devices. For example, Samsung Pay uses secret information in the application code to generate an encryption key for credit card information, which was revealed in [1]. KakaoTalk, a widely used messenger in Korea, encrypts and stores a list of friends and conversations in a database; nonetheless, the data stored in the KakaoTalk database was exposed through static analysis [2].

As the associations between mobile devices and daily activities, including business, have increased, the demand for remedies to securely store sensitive data of users has increased. However, most studies on protecting private information in Android have focused on the information generated by Android and not the sensitive information generated by third-party apps. From a user's perspective, the personal information produced by third parties must also be securely protected, and a systematic protection method for such information is required.

The proposed security architecture protects sensitive information in databases, including databases for third-party apps. FDE can not protect the database from exposure, because an attacker, who can access the database file, is able to see database content stored in plaintext. Furthermore, for a database system in the application domain, an attacker can acquire crucial information related to an encryption key by simply analyzing a restricted domain range only. To remedy these security breaches, the solution entirely separate the database management system from the application domain. The separated database management system encrypts a database with a key that is managed by itself. Plaintext no longer exists in the database, and no key material can be revealed by reverse engineering of the app domain. As a result, developers do not need to manage the security of databases in the process of developing apps, because the separated database system manages the security. This architecture covers not only default Android databases but also databases of third-party applications. It operates transparently on the current system and is robust even for the root attacker.

This solution proposes a one-to-one paring between the owner app and its database. To this end, this solution uses the runtime information of an app to obtain a fingerprint of the app, which is unique. This tight pairing prevents an app for accessing a database owned by other

app, even if it possesses the uid of the owner app

The security and performance of the proposed schemes are analyzed on a real device. The overhead introduced by the SQL operations in the architecture is measured. The changes in overhead caused by increasing the number of concurrent processes that execute the SQL operations are observed. The results show that the proposed security architecture provides functionalities to protect database access securely and stably, with acceptable overhead.

A method to check if the update process is correct, i.e., to check if the updated version is actually updated from a previous version is also proposed. Because an app could be updated frequently, security architectures such as this should carefully identify the app during an update process. The current system verifies the signature and the package name of the app before updating an app. If the developer is the same, the signature of apps are verified with the same certificate. The package name is simply an alias assigned by the developer. So more comparisons in the strict sense will be needed in the future to ensure that a new app is an update of a previous version. To this end, an app similarity comparison method is proposed, which can be processed in a local device. The proposed method could be utilized to strengthen the relationship between two apps (previous and updated app), and eventually, between an app and its database. To evaluate the feasibility of the comparison method, we collect a set of apps of different versions and calculate similarity scores for each pair

1.1 Preliminaries

To understand the solution, certain concepts/technologies must be introduced to the reader first and these are

SELinux and SEAndroid

SELinux [3] is an access control mechanism that works alongside with the Linux regular access controls. It is an instantiation of the FLASK architecture and is an implementation of the mandatory access controls (MAC) framework on Linux. SELinux enforces the access controls according to SELinux policy, which is loaded during system startup.

Each subject (process) and object (resource such as files, network resources, etc.) is labeled with a security context (also known as security label) that contains a type attribute. Most

of the rules are made of SELinux contexts and used to determine whether an access is allowed or not, as defined in the policy. An access must be granted through SELinux policy rules. An allow statement of an SELinux rule is structured as follows:
`allow domain type:class { permissions}` where domain is a subject type, type is an object type, class is an object class, and permissions are a set of operations. This statement means that a domain (process) is allowed to perform an operation in permissions with a specific function supported by class on a type (target resource).

SEAndroid [4] is an implementation of SELinux for the Android Linux Kernel. The main features distinguishing SEAndroid from SELinux are the components added to the framework and Android-specific policy. SEAndroid adds an Android-specific policy tailored for the Android eigen components. SEAndroid labels app processes according to the SELinux security context. However, at the time policies are created, it is not known a priori which application will be installed on the system. Therefore, SEAndroid decides the security contexts of an application at its install-time. Currently, all the third-party applications are assigned after they are installed, as a single SELinux context: `untrusted_app`.

Android Database (SQLite)

Unlike most other Database Management Systems (DBMS) that use a client-server model, SQLite [5] does not have a separated processing server, but is an in-process library that implements an embedded SQL database engine. That is, SQLite is embedded into a program and does not support a multi-user database. The SQLite database is stored in and managed with a single file. This implies that anyone with access to the database file can read all database content.

Android provides a built-in SQLite database implementation. Similar to other files stored on the device's internal storage, Android maintains a database in private space. Because of the well-known "Android Application Sandbox" characteristic and other security features, one application cannot directly access the resources of other applications. Each database is accessible only by the app that generated the database.

Android provides a way to share one app's database with other apps by using a characterful component, called a Content Provider. When a developer of an app wants to share the

app's database with other apps, the developer implements a Content Provider associated with the app by defining a Uniform Resource Identifier (URI) for the database. Android databases can be classified into two types: databases generated by third-party apps and databases generated by Android (such as contacts, calendar, and e-mail). Accessing a database of any type is managed by the corresponding Content Provider. However, access to a database generated by Android requires a permission defined in the Android Permission SetâĂŤacquiring permission is compelled by the Android Permission FrameworkâĂŤwhereas access to a database generated by a third-party app does not require any permission (though user-defined permissions do exist, it is an optional feature).

Trusted Execution Environment (TEE)

A Trusted Execution Environment [6], [7] is a secure area of the main processor. It is isolated from a Rich Execution Environment (REE) (e.g., mobile OS) to protect assets such as cryptographic keys and user credentials. The TEE provides security features such as isolated execution, integrity of Trusted Applications (authorized security software which is loaded and executed in TEE) along with confidentiality of their assets. An attacker in the REE cannot access resources of TEE directly. Any direct access from the REE to the TEE is restricted by a hardware access-control mechanism. In general, the TEE offers an execution area that provides a higher level of security than a REE (e.g., mobile OS) and more functionality than a Secure Element (SE). The wellknown example of a TEE for embedded devices is ARM TrustZone. It is currently utilized to provide security-critical services

Android Keystore

The Android KeyStore [8] stores cryptographic keys in a container to prevent the extraction of security materials from the device. The keys in the KeyStore can be used only through cryptographic operations while the keys are non-exportable. The KeyStore system may involve TEE [6], [7]. To use this hardware based secure storage, a manufacturer's support is needed. If TEE is not available, the KeyStore system is implemented by a software. The KeyStore system is used by the Android KeyStore Provider (referred to below as AndroidKeyStore), which was introduced in Android API level 18. AndroidKeyStore interacts with the KeyStore service that interacts with Keymaster in TEE.

When an application generates a key using AndroidKeyStore, the key is stored in KeyStore under an alias. This alias is a string value defined by the app developer, and is used to identify the corresponding key. Keymaster is responsible for the cryptographic operations with keys linked to each Android app. However, the keys do not reside in the TEE area; they are stored in the files in the `/data/misc/keystore/user_0` directory. These files are encrypted using a device-specific key in the TEE, of course, and their names are correlated with the user ID of the app and the app developer-defined alias

Existing system

Current Android controls database access based on the Linux uid of an app. It is possible for a developer to make two applications share the same uid so that the two applications are able to access each other's database files. While this sharing method is helpful to conserve system resources, it may cause a security and privacy flaw. It is difficult for users to know which apps have the same uid; moreover, users have no controls over this arrangement. In addition, an attacker with root privileges can manipulate the uid to gain access to a target database.

As SELinux has become a distinguished security solution to control access using MAC, numerous studies has focused on database protection based on SELinux. For example, SeSQLite [9] provides additional access controls based on SELinux security policy. In particular, SeSQLite demonstrated that access control with minimal overhead on performance and size by integrating MAC into Android builtin SQLite is feasible. Its access control methods can control access through the corresponding DBMS. However, it can not prevent database content from being exposed to someone who can directly access the database file. Most database formats are already well known, so anyone with a database file can easily view the stored content. SeSQLite assumes that an Android app has its unique SELinux contexts and intends to make policy using this information for access control. However, it is difficult to configure the system in a manner that allows all apps to automatically have individual contexts at install-time. In fact, to the contrary, all third-party apps are assigned to a single SELinux context: `untrusted_app`.

Because FDE can not protect data during operation, all apps must individually prepare countermeasures to protect their data during operation; for example, external libraries such as SQLCipher [10] can be used. However, this type of remedy might limit interoperability because its configuration is only compatible within its domain. To construct a secure area isolated from an unsecured domain, Samsung KNOX introduced a secured container on their Androidbased device. In the secured container, only approved apps can be executed and their data can be utilized within a secured domain only. Because of this, however, only a few selected apps can have the benefit of this protection.

Android KeyStore provides functionalities that protect key material from unauthorized

use. Unfortunately, several vulnerabilities have been found. The work of analyzed KeyStore with various options. From their results, even if KeyStore uses TEE, an attacker can use the keys of other app on the device by renaming the related files with the proper uid. The work of discovered a problem in which the files generated by KeyStore exist even after the owner app is removed. Because of this, if a new app has the same uid as the removed app, the new app can use the keys of the removed app. However, this vulnerability was removed by Google.

The main goal of an attacker is to disclose informati on in a database or manipulate database content without being detected

one app cannot directly access the resources of other apps under normal circumstances. The target database of an attacker belongs to a specific app (e.g., system or third-party app). Thus, the attacker needs some level of privilege to access the database. The soltuion assume that an attacker is able to acquire this capability by performing software attacks on the Android software stack.

In order to gain a permission or escalate the privileges for database access, an attacker attempts attacks on the higher layer of the Android software stack; attack types may include malicious apps, confused deputy attacks, and collusion attacks. In addition, we assume that the attacker will launch attacks on the lower layer of the Android software stack. Previous studies have described various privilege escalation attacks on the lower layer of the Android software stack. These attacks grant an attacker root privileges. A root attacker with root credentials can run apps with root privileges; in this case, all available permissions are inherited and the file system can be inspected at will. SELinux and SEAndroid have been included in recent Android versions; however, a root attacker can access the entire file system

An attacker is also assumed to have the ability to decompile and analyze Android Package Kit (APK) files to obtain information related to database operations. However, the attacker can not manipulate the memory space of the target process, which is protected by SELinux. Protecting the manipulation of the memory space can be achieved by applying an antidebugging technique to the target process or limiting the execution of memory access tools and/or systemcalls with SELinux policy.

The solution assume an attacker cannot compromise the kernel (and bootloader also).

An attack that compromises the kernel can take control of the entire system. However, there are several technologies designed to protect the kernel from malicious attacks. Secure Boot (also known as Trusted Boot) guarantees the proper bootloader and kernel were loaded and run when the device was started. Real-time Kernel Protection protects the kernel from attacks that exploit an already booted and running kernel. So, a kind of ”Evil Maid” attack is not feasible for attackers.

The KeyStore operates with a TEE. If a TEE is not available, the KeyStore system operates with a software implementation; however, we do not consider such software-based environments in the architecture.

The solution assumes that there are no vulnerabilities in the added components. The extent of vulnerabilities (the number of defects) is relevant to the complexity of software. The solution's components are functionally simple and have a relatively small amount of code, thus software vulnerabilities can be effectively removed through static and dynamic analyses. As a result, we assume there is no chance for attackers to acquire the privileges that the components have

The memory space of a running app is not in the protection scope of the architecture. For an app to use the data stored in the database, the data must finally exist in the memory of the app in plaintext form. Thus, we do not consider an attacker who attempts to access the memory space of a running app. In a similar sense, data that are shared through a Content Provider are also not protected. If an app has permission to access a specific Content Provider, it can access database content that the owner wants to share. In a Content Provider scenario, we note that the database transaction is actually performed by the owner app. Thus, in a situation where the owner of the data has granted the access path through a Content Provider, we left the security of data provided by a Content Provider to the Android Permission Framework.

Proposed system

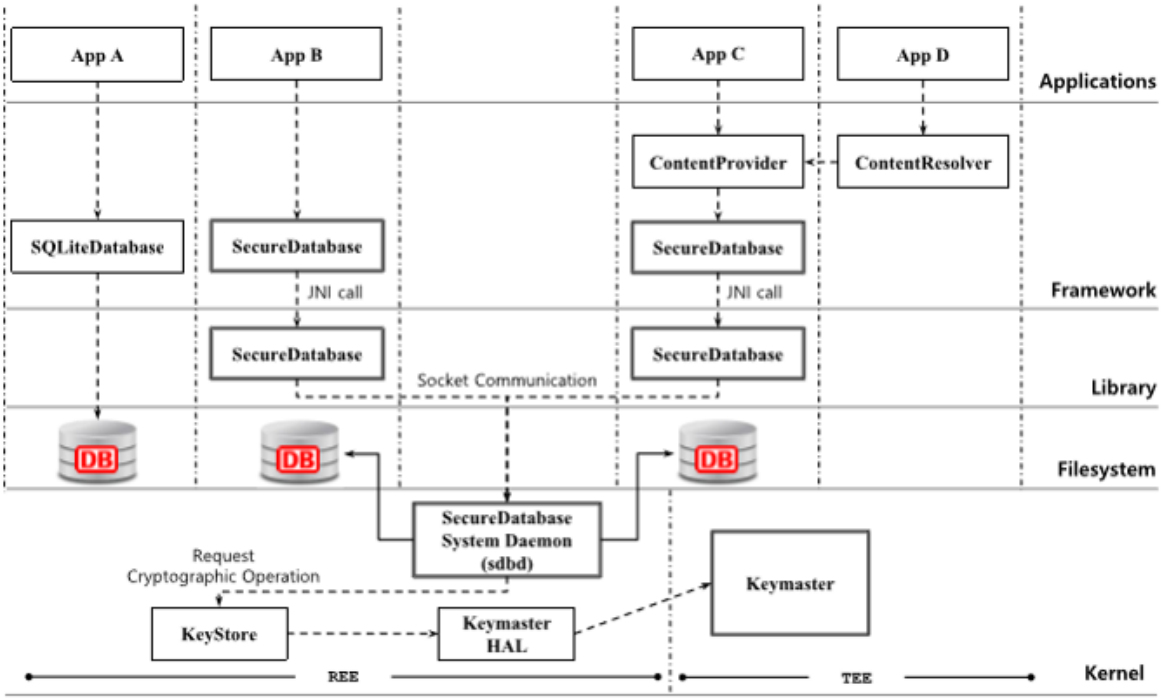


Fig. 3.1: System Architecture Overview (IEEE Access, January 29, 2018 - Security Architecture for a Secure Android Database)

Fig. 3.1 represents an overview of the architecture. Unlike current SQLite database of Android, the database engine is not embedded into a specific app but rather into the native daemon, which is named sdbd (SecureDB Daemon). All the database transactions are performed by sdbd. To that end, related classes have been added to the Android framework, in order to pass requests generated by an app to sdbd and receive the results. sdbd acquires information of the connected app through data provided by the kernel. Based on this information, it creates and manages the database on the specified path of the app.

This approach basically provides database confidentiality by using encryption. At the first initial booting time, sdbd requests a key generation to KeyStore. KeyStore generates a key (K_{sdbd}) for sdbd and encrypts K_{sdbd} using a device-specific key (Kdevice) stored in the TEE. sdbd randomly generates an encryption key $K_{databaseapp}$ for each app and database pair, and

$K_{databaseapp}$ is encrypted using K_{sdbd}. Hence, the key hierarchy in the proposed architecture consists of three layers. The encrypted $K_{databaseapp}$ is stored in a meta-data file by sdbd.

The resources managed by sdbd are protected by SELinux policy so that only a privileged subject (i.e., sdbd) can access them.

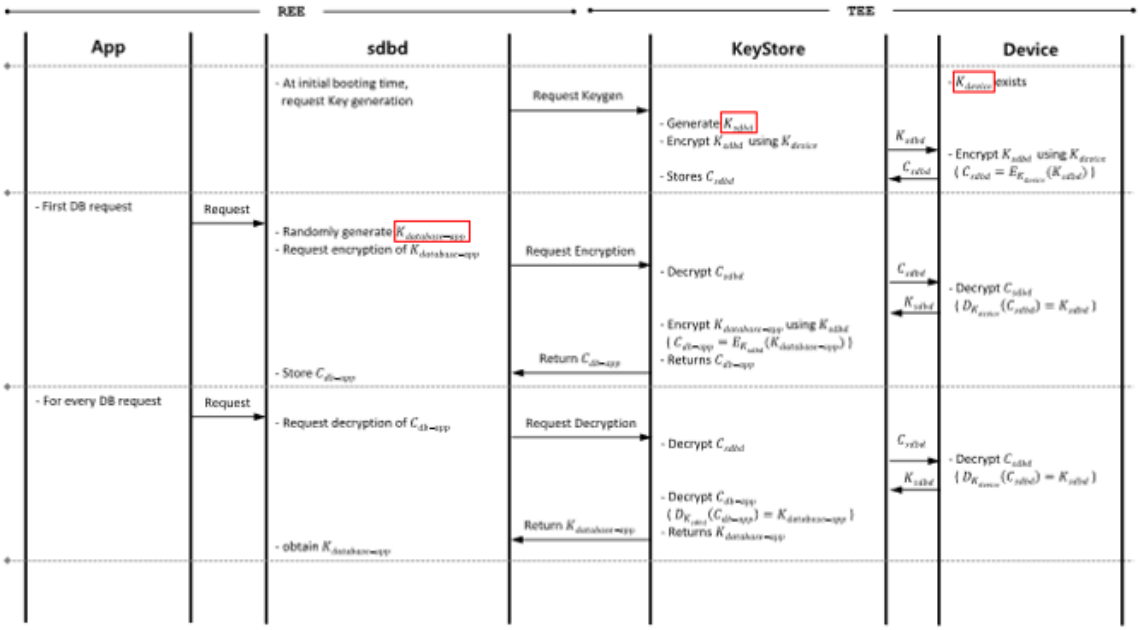


Fig. 3.2: Key generation overview of the proposed secure database architecture and the sequence of key decryption/encryption for each database request. Ek (m) denotes the encryption of message m using key k, and Dk (c) denotes the decryption of ciphertext c using key k.

Whenever an app asks sdbd to perform a database transaction, sdbd authenticates the app. By verifying the app that requests a transaction, sdbd allows only an authorized app to access its database (App-binding). After app authentication, sdbd requests KeyStore to decrypt the encrypted $K_{databaseapp}$ in order to obtain $K_{databaseapp}$. At this time, KeyStore also authenticates sdbd. If KeyStore verifies sdbd correctly, KeyStore decrypts the encrypted K_{sdbd} using K_{device} in the TEE. Thereafter, sdbd can perform a cryptographic operation through KeyStore while the related key (K_{sdbd}) is unknown to sdbd. In other words, K_{sdbd} can only be used by sdbd, on the same device that has a corresponding device-specific key K_{device} (Device-binding). Finally, sdbd performs the requested transaction using $K_{databaseapp}$ on the encrypted database and sends the result to the app. The key hierarchy procedure is depicted in Fig. 3.2.

The proposed architecture also proposes fine-grained access control over a database and KeyStore. The current Android uses uid for access control over a database and KeyStore, which might be insecure because multiple apps may have a single uid or an attacker might manipulate the uid. In this architecture, sdbd authenticates an app via the fingerprint of the app, and KeyStore authenticates sdbd via the fingerprint of sdbd. Such a fingerprint is unique information, which is generated using the runtime information of the object to be authenticated.

In the following, the individual components that comprise the secure database architecture are explained (see Fig. 3.3).

3.1 SELinux policy

The solution leverage SELinux to protect the components and resources from unauthorized access. Although Android leverages the Linux kernel and SELinux is a security module available in the standard Linux kernel, SELinux has been modified specifically for Android.

The main difference is in the security label. Each process or object is labeled with a security label. In conventional SELinux, a security label takes the form of `[user:role:type:mls_level]` consisting of four different fields. However, only two fields, role and type, are actually used in AOSP (Android Open Source Project). The other fields are fixed; user is set to 'u' and `mls_context` to 's0'. The field role is always set to 'r' for domains (processes) or `'object_r'` for objects. Thus, the type is the primary field for the access decisions.

In accordance with the SELinux implementation of Android, we define types for our components: 'sdbd' for domain, '`sdbd_data_file`' and '`sdbd_exec`' for object. The solution specifies our domain, type definitions, and rules in the 'securedb.te' file. The rules are not complex. They are defined by 'allow' and 'neverallow' rules, which means that only the sdbd domain (some rules include 'init') can access (proper operation) objects that are labeled with '`sdbd_dat_file`' or '`sdb_exec`'. Additionally, rules for the use of KeyStore and Binder, and for limiting access with system-calls (such as ptrace) are defined. Several definition files (e.g., `file_contexts`, `service_contexts`, init.te, etc.) related to the operation of the components are also updated.

In addition, because sdbd communicates with PackageManager using the FILESYSTEM socket of UDS (Unix Domain Socket), we also define rules that allow only sdbd and `system_server`-PackageManager is executed in `system_server` processâĂŤto access the socket file.

3.2 Keystore

The current Android KeyStore verifies the ownership of keys using an app's `uid`; thus, an app typically can not access other app's key. When an app attempts to access KeyStore by using an `uid`, KeyStore grants an access to all the keys corresponding to the `uid`. Because the root attacker can manipulate an `uid`, it is not secure for KeyStore to grant access to keys only using a `uid`.

In the proposed architecture, KeyStore generates and stores a cryptographic key for `sdbd` in the TEE area. To ensure that this key is only used upon a request from `sdbd`, KeyStore verifies the identity of the caller process (i.e., `sdbd`). As we mentioned above, a root attacker can control an `uid` and can request KeyStore to perform a cryptographic operation with a fabricated `uid`, e.g., the `uid` of `sdbd`. To prevent this attack, we modify KeyStore such that the trustlet ($Keymaster_{new}$) of KeyStore consists of three modules: Mode Decision, Caller Authenticator, and $Keymaster_{old}$ as shown in the right-hand box of Fig. 3.3. $Keymaster_{old}$ denotes the Keymaster of the current Android.

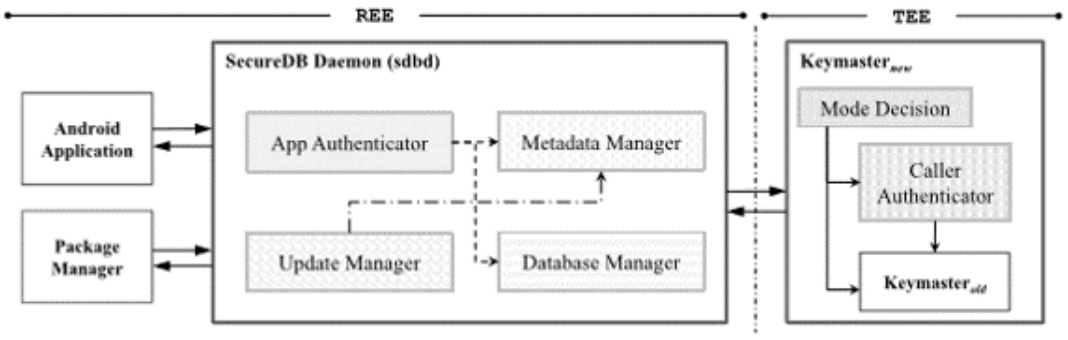


Fig. 3.3: Simple abstract model of the design

For branching the way of Keymaster operation, we add a flag parameter that is uti-

lized when KeyStore calls $Keymaster_{new}$. Depending on this flag, the Mode Decision of $Keymaster_{new}$ proceeds to Caller Authenticator or proceeds to $Keymaster_{old}$. That is, if KeyStore perceives the caller as sdbd, KeyStore sets the flag to authenticate the caller process, and $Keymaster_{new}$ proceeds to the Caller Authenticator module according to the Mode Decision. Otherwise, $Keymaster_{new}$ directly proceeds to $Keymaster_{old}$; in that case, it operates in the same manner as the current Android Keymaster.

This module authenticates that the caller process is sdbd. To that end, we utilize a hash value of runtime information, which is related to a portion of sdbd, as a fingerprint. In more detail, the hash value is computed using a code segment that contains sections holding machine instructions as well as sections holding certain read-only data. Accordingly, unless sdbd is not modified, the code segment of sdbd remains unchanged. The fingerprint (hash value) is stored within an encrypted file that is managed by KeyStore.

Because KeyStore is able to know the pid of a caller process by itself, KeyStore passes the pid of the caller process to $Keymaster_{new}$. Each time sdbd requests KeyStore to perform a cryptographic operation, Caller Authenticator accesses the memory space of the running process that matches to the pid and reads the code segment of the process. After calculating the hash value, Caller Authenticator checks if the hash value is identical to the stored fingerprint of sdbd. Only if two values (the calculated hash value and the stored fingerprint) are identical, the Caller Authenticator proceeds to $Keymaster_{old}$. $Keymaster_{old}$ then performs the requested cryptographic operation.

3.3 SecureDB daemon

The SecureDB daemon (denoted as sdbd) is the main component in the architecture. It is started by init when the device boots. sdbd is a system daemon and it communicates with Android apps and PackageManager using UDS. sdbd consist of four modules: App Authenticator, Meta-data Manager, Update Manager, and Database Manager

This module authenticates an app connected to sdbd. When a connection is initially established between an app and sdbd, sdbd verifies the identity of the app. Please recall that

the code segment of sdbd is used when KeyStore authenticates sdbd. However, sdbd can not authenticate the connected app in the same manner.

Unlike programs on most other systems, Android apps do not have a single entry point (such as a main function); moreover, the class is loaded by a class loader at runtime if necessary, so we cannot guarantee that a portion of the process memory will be consistent at a specific point of time. To authenticate an app in the proposed architecture, sdbd uses all of the executable format files (e.g., dex, odex, or oat files) and APK files, which are referenced by the app process. sdbd is able to know the pid of the connected app process by itself. Thus, using information provided by the kernel, sdbd can find the files referred to by the process that matches to the pid. sdbd calculates a hash value using the above-mentioned files and uses the hash value as a fingerprint of the app. Every app process should reference at least its own executable, and such structure is managed by the kernel. The solution assumes that the kernel is not compromised, thus, such reference information is reliable. Because such a fingerprint is different for each app, it is used as an identification information of an app.

sdbd also checks that the caller is Android friendly, not a dummy created by an attacker; attackers who know the mechanism of App Authenticator attempt to create a fake process to defeat the verification procedure. The solution check whether the caller is Android-friendly by inspecting that the caller is a child process of zygote and that the memory is configured accordingly.

Only if all verifications by the App Authenticator are successful, a requested database transaction is performed by the Database Manager.

This module manages a meta-data file, which is related to the target application and its database. A meta-data file stores all data used for app authentication and database operations. This file is named in the following form in a specific path that is managed by sdbd: `[APP_UID]_[APP_PACKAGE_NAME].key`. A meta-data file is generated by sdbd when an app requests opening a nonexistent database for the first time.

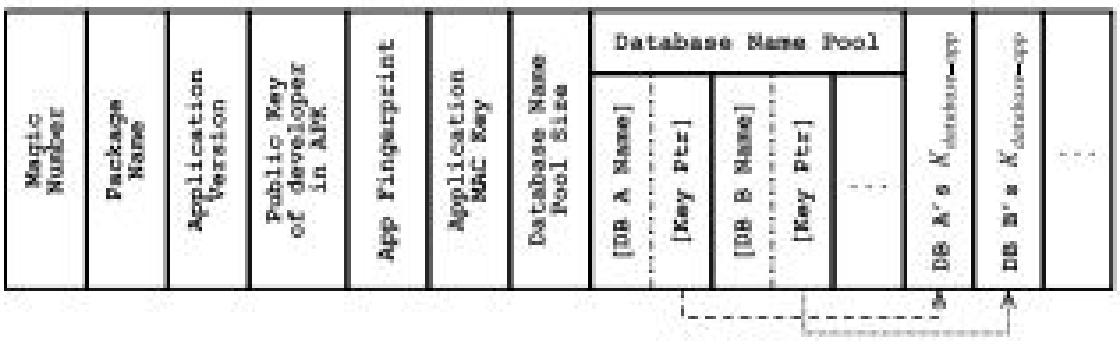


Fig. 3.4: structure of metadata file

Fig. 3.4 shows a structure of a meta-data(.key) file. The $K_{database_app}$, which is used for database encryption, is randomly generated by sdbd and the fingerprint of app is generated as described in Section V-D1. After all the content for the meta-data(.key) file is ready, it is finally encrypted by KeyStore with a key (K_{sdbd}) preserved in TEE.

This module is in charge of database operations and communications with an app. When an app requests a database operation and if a corresponding meta-data file exists, the App Authenticator Module compares the fingerprint of the requesting app process with the stored fingerprint. If the authentication is successful, sdbd and the app share a random session key (SK) using the Diffie-Hellman method. Afterward, the entire communication is encrypted by the AES-CTR algorithm using SK. To perform operations on the database, an app generates an SQL query and transmits it to sdbd. sdbd performs the database transaction and returns the result to the app. If the query is a SELECT statement, a SecureCursorWindow is created, similar to a CursorWindow created by the Android database Cursor. The SecureCursowWindow is passed to the app as a SecureCursor, which manages the SecureCursorWindow. Unlike CursorWindow, the content of the SecureCursorWindow is encrypted using SK; thus, it can be decrypted only while the session is maintained.

A database key (DBKey) is required to perform transactions on an encrypted database. sdbd optionally supports user consent (e.g. password) for generating a DBKey. If user consent is not available, sdbd decrypts the metadata(.key) file through KeyStore to obtain $K_{database_app}$ and uses $K_{database_app}$ as a DBKey to work with the corresponding database. Otherwise, if user

consent is available, the USR_KEY is derived from the user consent using the Key Derivation Function (KDF). The DBKey is obtained by XOR of $K_{database_app}$ and USR_KEY.

Unlike the assumption that an update of a well-structured sdbd is not required, updates of apps occur frequently. If an app is updated, the fingerprint of the app, which is used for authentication, will be changed. Thus, sdbd must renew the stored fingerprint in a meta-data file while an app is updated.

sdbd opens two types of sockets: one for apps, and one for PackageManager. The socket for apps is used to deal with database requests of an app. The socket for PackageManager is used to inform the update status for each other. While PackageManager updates an app, PackageManager connects to sdbd and informs which app is being updated.

Using a two-step process, sdbd checks the old app and the new app to determine whether it should allow an update of the stored fingerprint. First, sdbd checks through the Manifest file to determine whether the new app intends to use a secure database (permission) and whether the new app is an updated version of the old app (Package name and Version). Second, sdbd verifies that the two apps are signed with the same key. The signature and the Manifest file are already checked by PackageManager; however, sdbd autonomously checks them again. Because Android manages application information (including the certificate) in several files (e.g., packages.xml, packages.list), an attacker may attempt to update/replace the current app with another app regardless of the app signature by modifying such files and/or filesystem information. If sdbd is configured to receive a check result from PackageManager, the result can be compromised by attackers. Thus, in the architecture, sdbd and PackageManager operate independently of each other and only the information indicating the target app is passed from PackageManager to sdbd.

The update process of an app's fingerprint by sdbd is processed after the updated app already installed, so sdbd uses stored data in a meta-data file as information of the old app. After the signature and the Manifest check succeed, Update Manager updates the stored fingerprint and instructs PackageManager to continue its process.

3.4 Interface and permission

The solution implement APIs for database transaction that are mostly identical to the current Android SQLite APIs. The changed part is an 'open' API that has an added parameter, which is used to input a user-consent (e.g., user password). The secure database can work without user-consent, but user-consent provides stronger security.

Further, A permission for secure database named "ACCESS_SECURE_DATABASE" is added on Android Permission Set. Currently in Android, an app can access its owned database without permission. Permission is required to access any database that an app does not own. However, the permission is added for the purpose of privileges that handle access to the owned secure database. In Android, system permissions are divided into normal permissions and dangerous permissions depending on whether an app needs to access a user's private information. Existing permissions for accessing Android's default database(e.g., READ_CALENDAR, READ_CONTACTS, etc.) are classified as dangerous permissions; however, the permission is classified as a normal permission. The database content that is stored by an app might contain sensitive private data, such as that contained in Android's default database. However, whereas accessing Android's default database is a type in which a third-party app accesses a database created and managed by the system app, a secure database is a database that is created and managed by an app itself; thus, the app has ownership of the database.

The solution's permission has two roles; First, when PackageManager and sdbd check the Manifest during the app update phase, the permission instructs them to follow procedures for secure database. Second, the permission informs the user that the app uses a secure database. As a result, the developer can configure the app to use the secure database in a manner similar to the way it used the existing database, through a minimally changed API. Moreover, after permission is granted, the user can be assured that their data are securely stored via a secure database.

3.5 Operation flow

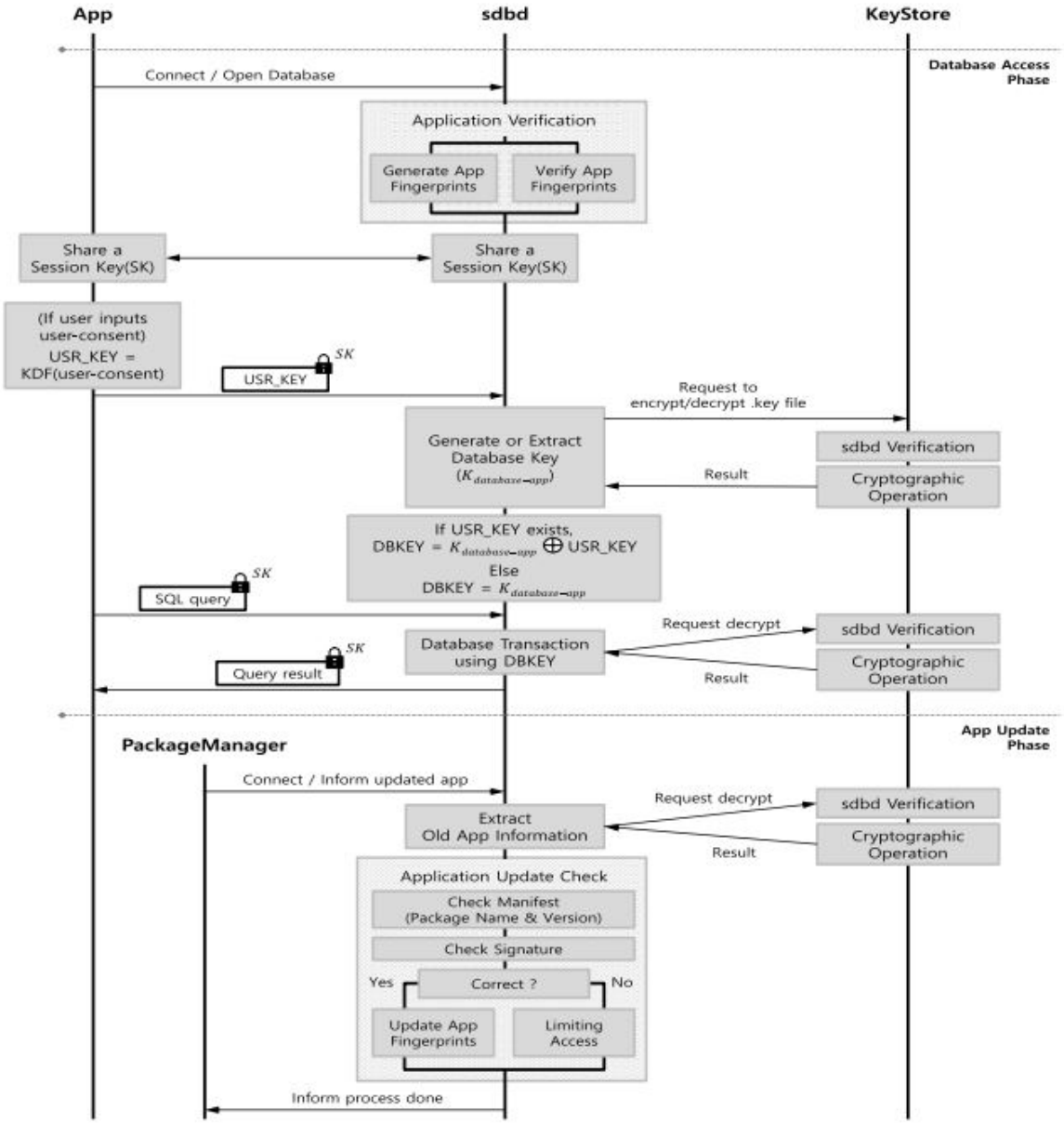


Fig. 3.5: Overview of operation flow

As shown in Fig. 3.5, the operation of the proposed architecture is divided into two phases: a Database Access Phase and an App Update Phase. Because the details of each entity's operations are already described in the sections for the individual components, this

section briefly presents the sequence of operations to be performed in the architecture.

In the database access phase, an app attempts to access a database and requests queries to be executed over the database. At first, the app calls the open database API with a database name through the SecureDatabase class. The SecureDatabase classâĂŤbecause the instance of SecureDatabase class is included in the app, all app-side entries are hereinafter referred to as 'app'âĂŤtries to connect to sdbd internally. After the connection is established, the app sends a message to open a specific database. At this time, if this is the app's first request, sdbd generates a fingerprint of the app and stores related information in a metadata file. If a previous request from the app exists, sdbd verifies the fingerprint of the app. After checking whether the fingerprint match, sdbd and the app share a random session key (SK) using the Diffie-Hellman method. If the app receives a user-consent, it generates a `USR_KEY` using KDF and encrypts the `USR_KEY` using SK. It then sends the encrypted `USR_KEY` to sdbd. In the case of a first-time access request, sdbd generates a random key ($K_{database_app}$) and stores it to the meta-data file after encryption through KeyStore. Otherwise, sdbd decrypts the existing meta-data file through KeyStore and extracts $K_{database_app}$ corresponding to the target database. For every request to KeyStore, KeyStore authenticates sdbd. sdbd prepares a DBKey using the XOR of a `USR_KEY` and $K_{database_app}$ if `USR_KEY` exists; otherwise, it uses $K_{database_app}$ as a DBKey.

sdbd does not maintain the DBKey, as it is in memory. When one transaction is completed, sdbd removes the DBKey from its memory. sdbd maintains the DBKey in encrypted form through KeyStore. When the app requests an SQL query, sdbd restores the DBKey, performs the database transaction, and returns the query result to the app.

The app update phase is initiated by a notification from PackageManager. If the app that is to be updated declared a SecureDatabase permission in its Manifest, PackageManager connects to sdbd before finishing the update and informs sdbd which app is being updated. sdbd verifies that the new app is an update of the old app as described in Section V-D4. If the verification is successful, sdbd informs PackageManager to continue its process and replaces the old fingerprint with the new fingerprint. Otherwise, sdbd sets a flag indicating that the access is restricted.

3.6 Application similarity

In the current system, Android verifies the signature and the package name of the two apps (updated app and installed app) before updating. However, if the developer is the same, the signatures of the apps are verified using the same certificate. Moreover, the package name is simply an alias assigned by the developer.

It is believed that there might be several cases which suffer from current method and that current method is insufficient to identify an app in a strict manner. For example, a developer might want to change the package name of the published app. This can occur when the developer finds that the package name was mistyped, or when the client that ordered the app requests a change to a package name after the app is already live on the market. Previously, because the developer is the same, there are several techniques (e.g., same UID) that allow the new app to control the old app's database, which may already contain a significant amount of data. However, because the architecture only allows the owner app of the database to access its database, the new app can not access the database even if the app only changes its package name. In one more example, a developer publishes an app (denoted as A) and later revokes the publication of A from the market. Finally, he develops another app (denoted as B) that has the same package name as app A. In this case, A can be updated to B and B can access the database of A, even if the body of the app is completely different.

With consideration of the above cases, a pilot method is proposed that compares similarity between two apps. The entire process of the proposed method proceeds in two steps. The first step checks the meta information of the two apps. It checks the sequence of the version between two apps, and verifies that the apps are signed with the same key. The second step calculates a similarity score between two apps, and decides whether the newer app is an updated version of the older app. This procedure will take a place in an update operation, so it is required to apply a mechanism that is completed within an acceptable time. The idea that the code bases of the original app and the updated app may be similar was what motivated us to check app similarity. Owing to the nature of an app's structure, the resources (e.g., images, layouts, audio, etc.) can be changed entirely. Therefore, the instruction codes inside an app is

focused upon. The solution collect instruction sequences from an app and then utilize them for footprint generation. The footprint is used to compare the similarity between two apps. From now on, the proposed procedure is explained with an assumption that the similarity method is integrated into the database architecture.

Each Android app contains a few dex file, which contains the actual Dalvik bytecode for execution. The Dalvik bytecode consists of opcodes and operands; however, only the opcodes (two-byte hex values) is collected on a method-bymethod basis in each class. Afterward, instruction sequences is added to the BloomFilter. A BloomFilter is a probabilistic data structure that is used to test whether an element is a member of a set. While it is a space-efficient data structure, there exists the probability of false positives. The factors that affect the probability of false positives are the length of the BloomFilter (m), the number of hash functions (k), and the number of elements in a set (n). Under the same conditions, the longer the BloomFilter length and the smaller the number of elements, the lower the rate of false positives.

Each class has a different number of methods. Thus, the use of a fixed-length BloomFilter for an app may create space inefficiencies or result in a greater-than-expected rate of false positives. The length of the BloomFilter should be determined with consideration for both the rate of false positives and the number of elements

The probability of false positives F is determined as follows:

$$F = \left(1 - (1 - \frac{1}{m})^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k \qquad \text{(Equ: 2.1)}$$

Thus, the length of the BloomFilter (m) can be represented with the following formula:

$$m - \frac{nk}{\log(1 - (F)^{\frac{1}{k}}} \qquad \text{(Equ: 2.2)}$$

Fixing the false positive rate F at 0.001 and the number of hash functions k at 4, the length of BloomFilter (m) is determined by the number of elements n. Some classes in an app may have very few methods and the updated version of a corresponding class may have a significantly increased number of methods. In this case, the false positive rate of BloomFilter might be significantly increased, because the BloomFilter length is not sufficient. Thus,the

minimal number of methods in a class is set at 50. In addition, the BloomFilter length is set to be that corresponding to 1.5 times the actual number of methods, taking into consideration the false positives that might occur when methods are added in the class of the updated app.

To check the similarity, the similarity scores of the original app and the updated app has to be calculated. The solution uses the Jaccard similarity coefficient method for measuring the similarity score. The intuition, from the perspective of code size, is that the size of the instruction sequences of the method seems equal to the proportion occupied by that method in the entire program. The differences between the original code and the updated code can be represented using three alteration types: 1) addition of elements (e.g., class or method), 2) deletion of elements, and 3) modification of elements. However, in the end, regardless of the alteration type, the alterations cause a gap between the two apps and the proportion of the gap represents the degree of changes to the app. That is, the proportion of the same instruction sequences between two apps can eventually represent a similarity between them.

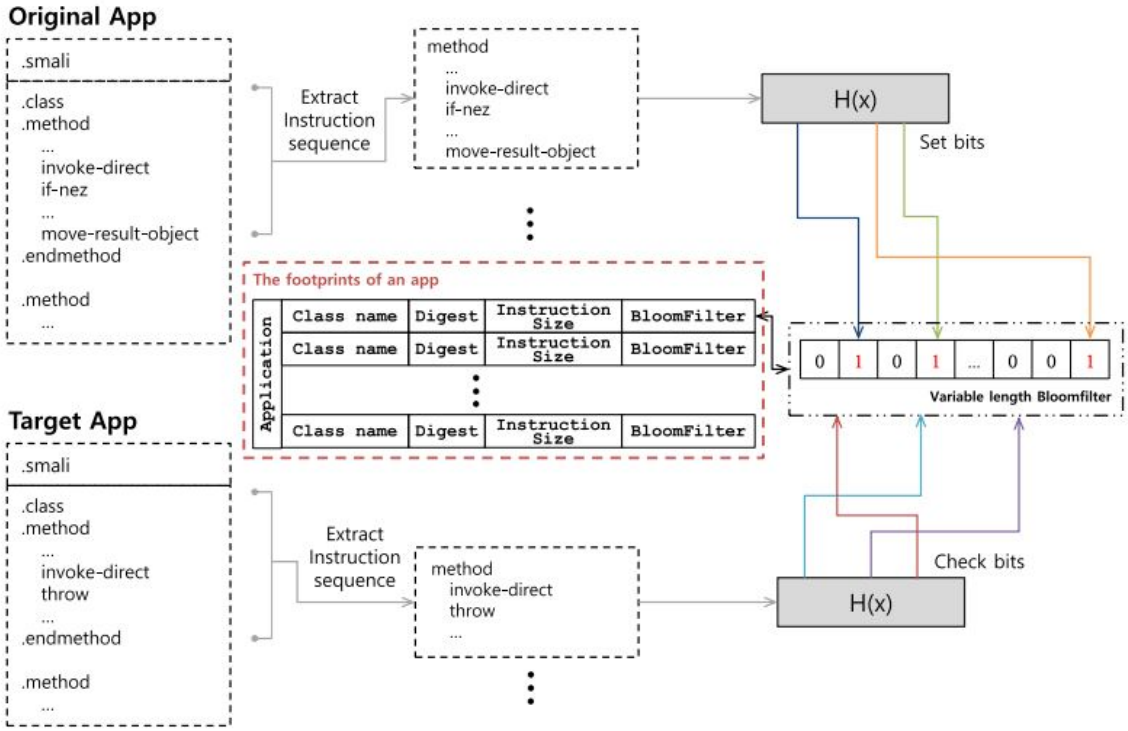


Fig. 3.6: Application Similarity Check Mechanism

The solution uses BloomFilter to check whether the same instruction sequences exist in

both the original app and the updated app. The proposed mechanism is shown in Fig. 9. The footprint of the original app is created when the app first attempts to access sdbd. sdbd extracts smali code from the app's dex files and collects instruction sequences for each method in a class. Then, to add an element to BloomFilter, sdbd feeds the instruction sequences to hash functions to obtain k array positions, and sets the bits at all these positions to 1. sdbd stores the BloomFilter in a meta-data(.fp) file in conjunction with the class name, digest of class, and the size of the instructions. A .fp file is not encrypted like .key file. Instead, sdbd calculates a MAC (Message Authentication Code) value using keyed-hash algorithms and a MAC key stored in the .key file; it then stores the MAC value at the end of the .fp file. Once an updated app is installed, sdbd extracts smali code from the updated app and collects instruction sequences for each method in a class again. sdbd gets a k array positions through keyed-hash functions and, if all of the bits at these positions are equal to 1, then it decides that the instruction sequences exist in the original app.

In the meantime, sdbd also calculates the size of instruction sequences, which is determined as the same, and computes the similarity score between two apps. To obtain the similarity score, sdbd computes the Jaccard similarity coefficient as follows.

$$SimilarityScore = J(ORI, UPD)$$

$$= \frac{\sum \mod M_{Ck^{ORI}} \bigcap \mod M_{Ck^{UPD}}}{\sum \mod M_{Ci^{ORI}} + \sum \mod M_{Cj^{UPD}} - \sum \mod M_{Ck^{ORI}} \bigcap \mod M_{Ck^{UPD}}} \quad \text{(Equ: 2.3)}$$

where $M_{Ci^{ORI}}$ is a set of methods that are composed of instruction sequences in class Ci (0 < i $\leq$ the number of classes in original app), $M_{Cj^{UPD}}$ is a set of methods that are composed of instruction sequences in class Cj (0 < j $\leq$ the number of classes in updated app), and Ck is a class in $\sum Ci \bigcup Cj (0 < k \leq$ the number of classes in $\sum Ci \bigcup \sum Cj)$.

3.7 Performance evaluation

This section summarizes several experimental results on the prototype implementations and explain the performance impact on the device. The measurements were performed on a LG Nexus 5X running a hexa-core 1.8 GHz processor with 2 GB of RAM. The device was loaded with AOSP 7.1.1(Build Number:N4F26I), which included the implementations.

SQL OPERATION First of all, to evaluate the performance impact on query processing, a series of experiments were executed comparing the implementations with both a base SQLite and a full database encryption solution named SQLCipher. SQLCipher is an extension to the SQLite library that provides transparent 256-bit AES encryption of database files. the AES-CBC algorithm with 128-bits key is used to encrypt a metadata file and generate a fingerprint with the SHA-256 hash function. In order to evaluate the overhead of the architecture that is independent of the database encryption strategies, SQLCipher is adopted for the database encryption solution.

SQLite provides a performance test suite named speedtest1 [53] to benchmark the library. Because SQLCipher is an extension of SQLite, it also provides a performance test suite [54]. However, such a test suite is not used, because the main focus of the experiments is not the performance of DBMS. As mentioned above, the architecture is scalable because it is independent of database encryption strategies. The concern is overhead caused by our architecture, and thus a new test suite and environment were built for the experiments.

In these implementations, the verification of KeyStore against sdbd is not involved, because it is very difficult to implement modified trustlet in TEE on real devices without the support of the manufacturer. It was taken into account that the KeyStore is not modified like the proposal in the experiments; hence, the operations of KeyStore is the same as the existing one. Instead, to reduce the gap between the proposal and the experiments, operations that read a memory and operate hash function with the same target (sdbd) were added in the implementation. The only difference is that the location of the operation is TEE in the proposal, but is REE in the implementation.

Operation	Target			Overhead
	SQLite	SQLCipher	Ours	
Open	22 ms	21 ms	376 ms	214 ms
CREATE	11 ms	472 ms	331 ms	
INSERT	10.3 ms	12.9 ms	23.2 ms	10.3 ms
SELECT	< 1 ms	< 1 ms	9.1 ms	≈ 9 ms
UPDATE	9.9 ms	12.8 ms	23.4 ms	10.6 ms
DELETE	11.1 ms	16.2 ms	26.7 ms	10.5 ms

Fig. 3.7: Performance comparison for basic SQL operations (avg. result of 10,000 executions)

We implemented an Android app to measure the performance of SQL at the app level. Table in Fig. 3.7 shows the overhead introduced by the basic SQL operations in our implementation. Compared with SQLCipher (we adopted SQLCipher for our database solution), the Open operation consumes about 350 ms more time. This overhead is caused by our authentication procedure, the app authentication of sdbd and caller authentication of KeyStore, and the key sharing procedure between an app and sdbd. Although this overhead is significant, Open occurs only one time when an app tries to access a database. For other operations, our proposed implementation requires a constant additional CPU time of about 10 ms. This overhead is caused by the caller authentication of KeyStore and secure communication through UDS. Basically, a single SQL transaction is a swift operation, so it seems that there exist considerable numerical differences in performance. However, the overhead, which is caused in our architecture, is a constant so we can enjoy security benefits with an additional time of about 10 ms. The mobile database environments do not deal with a heavy amount of transactions unlike a database servers; therefore, most users will not perceive the difference in performance between the existing database and ours.

The structural nature of our architecture overview may cause concerns about bottlenecks resulting from several processes attempting to simultaneously access the database. Thus, we performed a kind of stress-test. The CPU time of each method while increasing the number of processes that access the database is measured, and compared the variations of processing time

based on the the performance in the table in Fig. 3.7. Recent mobile devices support multitasking, but offers only limited multitasking that differs from traditional multitasking experienced in PC environments. It is considered that there might be only a few simultaneous tasks that will access the database, so experiments were performed while increasing the number of processes up to 10. The variations in processing time are within Âś 5ms. Compared with other methods, it can be seen that the variations caused by increases in the number of processes are within the overhead that occurs in multi-process environments (or within the error range of measurement), as in other methods. Therefore, the overhead caused by concurrent access can be considered as negligible

Conclusion

Android database is a well-structured data storage system, and therefore most apps utilize it as their private data warehouse. However, previous database systems in Android have been exposed to several risks that unexpectedly disclosed database content. App developers have to autonomously consider and respond to the security of their own stored data. This paper presents an architecture for a secure database environment on Android. The proposed architecture provides database confidentiality by allowing only the authorized entity to access the corresponding database. Only sdbd can access the unique key in TEE and only the authorized app can request a transaction to sdbd. Thus, with such a verification chain method, only the app with ownership rights over the data can access the database. The proposed method was designed to operate transparently on the current system and is friendly to developers and users with low performance overhead. Although this architecture was introduced on Android in this paper, it is expected that the approach can be applied to mobile OSes other than Android.

REFERENCES

[1] S. Mendoza, ”Samsung pay: Tokenized numbers, flaws and issues,” in Proc. Black Hat USA, 2016. [Online]. Available: https://www.blackhat.com/us-16/briefings.html#salvador-mendoza

[2] H. Kim, ”A security analysis of encrypted databases in KakaoTalk messenger,” in Proc. MSEC, Dec. 2016. [Online]. Available: http://www.msec.or.kr

[3] S. Smalley and P. Loscocco, ”Integrating flexible support for security policies into the Linux operating system,” in Proc. USENIX Annu. Tech. Conf. FREENIX Track, 2001, pp. 29âĂŞ-42.

[4] S. Smalley and R. Craig, ”Security enhanced (SE) Android: Bringing flexible MAC to Android,” in Proc. NDSS, vol. 310. 2013, pp. 20âĂŞ38.

[5] SQLiteâĂŤAn Embedded SQL Database Engine. Accessed: Nov. 2016. [Online]. Available: http://www.sqlite.org

[6] GlobalPlatform. (2017). TEE System Architecture Version 1.1. [Online]. Available: https://www.globalplatform.org/specificationsdevice.asp

[7] TPM MOBILE With Trusted Execution Environment for Comprehensive Mobile Device Security, TC Group, Risskov, Denmark, 2012.

[8] Android Keystore System. Accessed: Nov. 2017. [Online]. Available: https://developer.android.com/training/articles/keystore.html

[9] S. Mutti, E. Bacis, and S. Paraboschi, ”SeSQLite: Security enhanced SQLite: Mandatory access control for Android databases,” in Proc. 31st Annu. Comput. Secur. Appl. Conf., 2015, pp. 411âĂŞ-420.

[10] Zetetic LLC. Full Database Encryption for SQLite. Accessed: Dec. 2016. [Online]. Available: https://www.zetetic.net/sqlcipher/