International Journal of Safety and Security Engineering
Vol. 15, No. 12, December, 2025, pp. 2497-2510
Journal homepage: http://iieta.org/journals/ijsse

Secure Data Storage in Android: A DataStore-Based Encrypted Library

Abdennour Jebbar*, Ahmed El-Yahyaoui

IPSS Team, Computer Science Department, Faculty of Sciences, Mohammed V University in Rabat, Rabat 10000, Morocco

Corresponding Author Email: Abdennour_jebbar@um5.ac.ma

https://doi.org/10.18280/ijsse.151207

Received: 4 October 2025
Revised: 2 December 2025
Accepted: 18 December 2025
Available online: 31 December 2025

Keywords:
Android, EncryptedSharedPreferences, Jetpack, security, cryptography, XChaCha20, mobile

ABSTRACT

With the deprecation of EncryptedSharedPreferences, which was a widely used mechanism for securing local data storage on Android, applications in availability-critical domains such as telecommunications have faced the challenge of maintaining data-at-rest security. In this work, we analyzed the design and cryptographic choices of EncryptedSharedPreferences, identifying limitations related to algorithm selection and extensibility. Based on this analysis, we designed and implemented an alternative encrypted storage library built on Android DataStore, employing AES-256-SIV for key protection and XChaCha20-Poly1305 for value encryption. We evaluated the proposed approach through a security audit of a demonstration application, focusing on data-at-rest exposure via debug access, Android Package Kit (APK) backup and restore, and rooted device scenarios. The evaluation showed that sensitive data remained inaccessible in all tested attack models. Performance measurements indicated a significant overhead, ranging from one to two orders of magnitude (10× to 100×) compared to unencrypted DataStore operations, highlighting a clear trade-off between stronger cryptographic guarantees and runtime performance.

1. INTRODUCTION

Data has long been recognized as a critical asset, regardless of the medium on which it is stored. As the sensitivity of data increases, so does the need for mechanisms that ensure its confidentiality, integrity, and availability across its lifecycle, including storage, transmission, processing, and destruction. These requirements have historically been addressed through a combination of cryptographic techniques and controlled access mechanisms.

In the contemporary digital ecosystem, data generation and exchange have reached unprecedented scales. As of 2024, around 402.74 million terabytes of data are created daily [1]. This data is distributed across servers, personal devices, and cloud infrastructures, and is continuously exchanged over global networks. Modern digital devices—including smartphones, personal computers, and connected systems—store a wide range of information such as personal communications, medical records, authentication tokens, and user preferences. Although such data may not always appear sensitive from an end-user perspective, it can nevertheless be leveraged for profiling, targeted advertising, or more severe privacy violations if inadequately protected.

From a security standpoint, data is commonly classified according to three operational states: data in use, data in motion, and data at rest.

Given the volume and sensitivity of data handled on mobile devices, and the widespread deployment of Android across billions of devices, application developers frequently seek additional layers of protection beyond system-level encryption. To address this need, Android introduced EncryptedSharedPreferences in 2019 as part of the Jetpack Security library, providing application-level encryption for key-value storage using AES-based cryptography [2].

However, the deprecation of the androidx.security:security-crypto library in early 2024 has raised practical concerns regarding the continued availability of secure, officially supported mechanisms for protecting application data at rest. While some argue that such usage contradicts Google's architectural recommendations, the widespread adoption of EncryptedSharedPreferences demonstrates a persistent demand for secure local storage primitives, motivating the exploration of alternative solutions.

In this work, we address this gap by examining the limitations of EncryptedSharedPreferences and proposing a secure alternative built on top of Android's modern DataStore framework. The main contributions of this paper are as follows:

1. An analysis of the design and cryptographic foundations of EncryptedSharedPreferences and the implications of its deprecation.
2. The design and implementation of an encrypted DataStore-based storage solution leveraging AES-256-SIV and XChaCha20-Poly1305.
3. A security evaluation of the proposed solution against common data-at-rest and runtime attack scenarios.
4. A performance comparison with existing Android storage mechanisms to assess the practical trade-offs between security and efficiency.

2. ENCRYPTEDSHAREDPREFERENCES

2.1 Origins

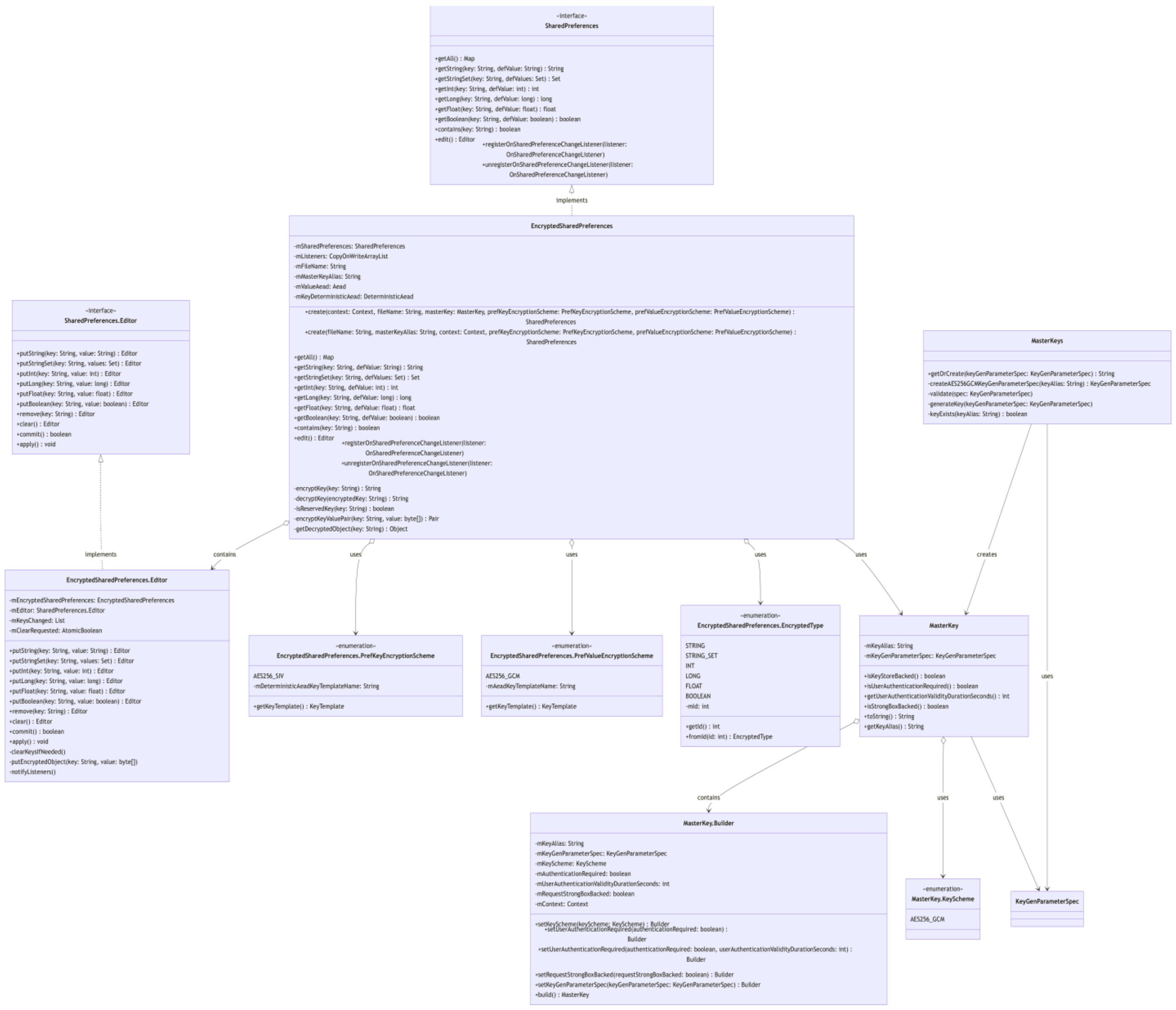
Since Application Programming Interface (API) level 1, Android contained a simple storage method called SharedPreferences. This method consists of storing key-value pairs in XML files. With its ease of setup and usage, many developers started using it more and more for saving small sets of data, session management, etc. [3].

With the evolution of apps, it started being used to handle more confidential data, such as usernames, emails, carts, etc. This required adding more encryption and brought to light "EncryptedSharedPreferences" with "Androidx Security Crypto 1.0.0" in February of 2020 [4].

2.2 Algorithm

As mentioned in the class diagram below (see Figure 1), EncryptedSharedPreferences implements the SharedPreferences interface, giving the user an API that is easy to handle and that he is used to. The sole difference between them is in the initialization, where the encrypted version takes the MasterKey Alias and the encryption schemes for keys and values. This class doesn't only use the classes cited in that diagram, but also other classes, ENUMs, types, etc., that are more specific to the implementation [5].

As for the MasterKey class, it is used to fetch the keystore path and save the alias in it to be used in operations afterward.



«interface»
SharedPreferences
+getAll() : Map
+getString(key: String, defValue: String) : String
+getStringSet(key: String, defValues: Set) : Set
+getInt(key: String, defValue: int) : int
+getLong(key: String, defValue: long) : long
+getFloat(key: String, defValue: float) : float
+getBoolean(key: String, defValue: boolean) : boolean
+contains(key: String) : boolean
+edit() : Editor
+registerOnSharedPreferenceChangeListener(listener: OnSharedPreferenceChangeListener)
+unregisterOnSharedPreferenceChangeListener(listener: OnSharedPreferenceChangeListener)
implements
EncryptedSharedPreferences
-mSharedPreferences: SharedPreferences
-mListeners: CopyOnWriteArrayList
-mFileName: String
-mMasterKeyAlias: String
-mValueAead: Aead
-mKeyDeterministicAead: DeterministicAead
+create(context: Context, fileName: String, masterKey: MasterKey, prefKeyEncryptionScheme: PrefKeyEncryptionScheme, prefValueEncryptionScheme: PrefValueEncryptionScheme) : SharedPreferences
+create(fileName: String, masterKeyAlias: String, context: Context, prefKeyEncryptionScheme: PrefKeyEncryptionScheme, prefValueEncryptionScheme: PrefValueEncryptionScheme) : SharedPreferences
+getAll() : Map
+getString(key: String, defValue: String) : String
+getStringSet(key: String, defValues: Set) : Set
+getInt(key: String, defValue: int) : int
+getLong(key: String, defValue: long) : long
+getFloat(key: String, defValue: float) : float
+getBoolean(key: String, defValue: boolean) : boolean
+contains(key: String) : boolean
+edit() : Editor
+registerOnSharedPreferenceChangeListener(listener: OnSharedPreferenceChangeListener)
+unregisterOnSharedPreferenceChangeListener(listener: OnSharedPreferenceChangeListener)
-encryptKey(key: String) : String
-decryptKey(encryptedKey: String) : String
-isReservedKey(key: String) : boolean
-encryptKeyValuePair(key: String, value: byte[]) : Pair
-getDecryptedObject(key: String) : Object
«interface»
SharedPreferences.Editor
+putString(key: String, value: String) : Editor
+putStringSet(key: String, values: Set) : Editor
+putInt(key: String, value: int) : Editor
+putLong(key: String, value: long) : Editor
+putFloat(key: String, value: float) : Editor
+putBoolean(key: String, value: boolean) : Editor
+remove(key: String) : Editor
+clear() : Editor
+commit() : boolean
+apply() : void
implements
contains
EncryptedSharedPreferences.Editor
-mEncryptedSharedPreferences: EncryptedSharedPreferences
-mEditor: SharedPreferences.Editor
-mKeysChanged: List
-mClearRequested: AtomicBoolean
+putString(key: String, value: String) : Editor
+putStringSet(key: String, values: Set) : Editor
+putInt(key: String, value: int) : Editor
+putLong(key: String, value: long) : Editor
+putFloat(key: String, value: float) : Editor
+putBoolean(key: String, value: boolean) : Editor
+remove(key: String) : Editor
+clear() : Editor
+commit() : boolean
+apply() : void
-clearKeysIfNeeded()
-putEncryptedObject(key: String, value: byte[])
-notifyListeners()
uses
«enumeration»
EncryptedSharedPreferences.PrefKeyEncryptionScheme
AES256_SIV
-mDeterministicAeadKeyTemplateName: String
+getKeyTemplate() : KeyTemplate
uses
«enumeration»
EncryptedSharedPreferences.PrefValueEncryptionScheme
AES256_GCM
-mAeadKeyTemplateName: String
+getKeyTemplate() : KeyTemplate
uses
«enumeration»
EncryptedSharedPreferences.EncryptedType
STRING
STRING_SET
INT
LONG
FLOAT
BOOLEAN
-mId: int
+getId() : int
+fromId(id: int) : EncryptedType
uses
MasterKeys
+getOrCreate(keyGenParameterSpec: KeyGenParameterSpec) : String
-createAES256GCMKeyGenParameterSpec(keyAlias: String) : KeyGenParameterSpec
-validate(spec: KeyGenParameterSpec)
-generateKey(keyGenParameterSpec: KeyGenParameterSpec)
-keyExists(keyAlias: String) : boolean
creates
uses
MasterKey
-mKeyAlias: String
-mKeyGenParameterSpec: KeyGenParameterSpec
+isKeyStoreBacked() : boolean
+isUserAuthenticationRequired() : boolean
+getUserAuthenticationValidityDurationSeconds() : int
+isStrongBoxBacked() : boolean
+toString() : String
+getKeyAlias() : String
contains
MasterKey.Builder
-mKeyAlias: String
-mKeyGenParameterSpec: KeyGenParameterSpec
-mKeyScheme: KeyScheme
-mAuthenticationRequired: boolean
-mUserAuthenticationValidityDurationSeconds: int
-mRequestStrongBoxBacked: boolean
-mContext: Context
+setKeyScheme(keyScheme: KeyScheme) : Builder
+setUserAuthenticationRequired(authenticationRequired: boolean) : Builder
+setUserAuthenticationRequired(authenticationRequired: boolean, userAuthenticationValidityDurationSeconds: int) : Builder
+setRequestStrongBoxBacked(requestStrongBoxBacked: boolean) : Builder
+setKeyGenParameterSpec(keyGenParameterSpec: KeyGenParameterSpec) : Builder
+build() : MasterKey
uses
uses
«enumeration»
MasterKey.KeyScheme
AES256_GCM
KeyGenParameterSpec

Figure 1. Simplified class diagram of EncryptedSharedPreferences class and its dependencies

2.2.1 Writing in EncryptedSharedPreferences

The sequence diagram in Figure 2 provides a macroscopic view of how data is stored, starting from the application creating the MasterKey before initializing EncryptedSharedPreferences. When needing to save data, the editor is fetched, passing to its methods (putString, putInt, …) our key and value. The Editor then proceeds to encrypt the data, key, and a concatenation of type id and value, then passes it through the parent SharedPreferences.Editor to save it in the file system through the apply method [5].

For this purpose, the key is encrypted in our key-value pair using the AES-256 SIV, which is a deterministic algorithm [6], and the value is encrypted using AES-256-GCM.

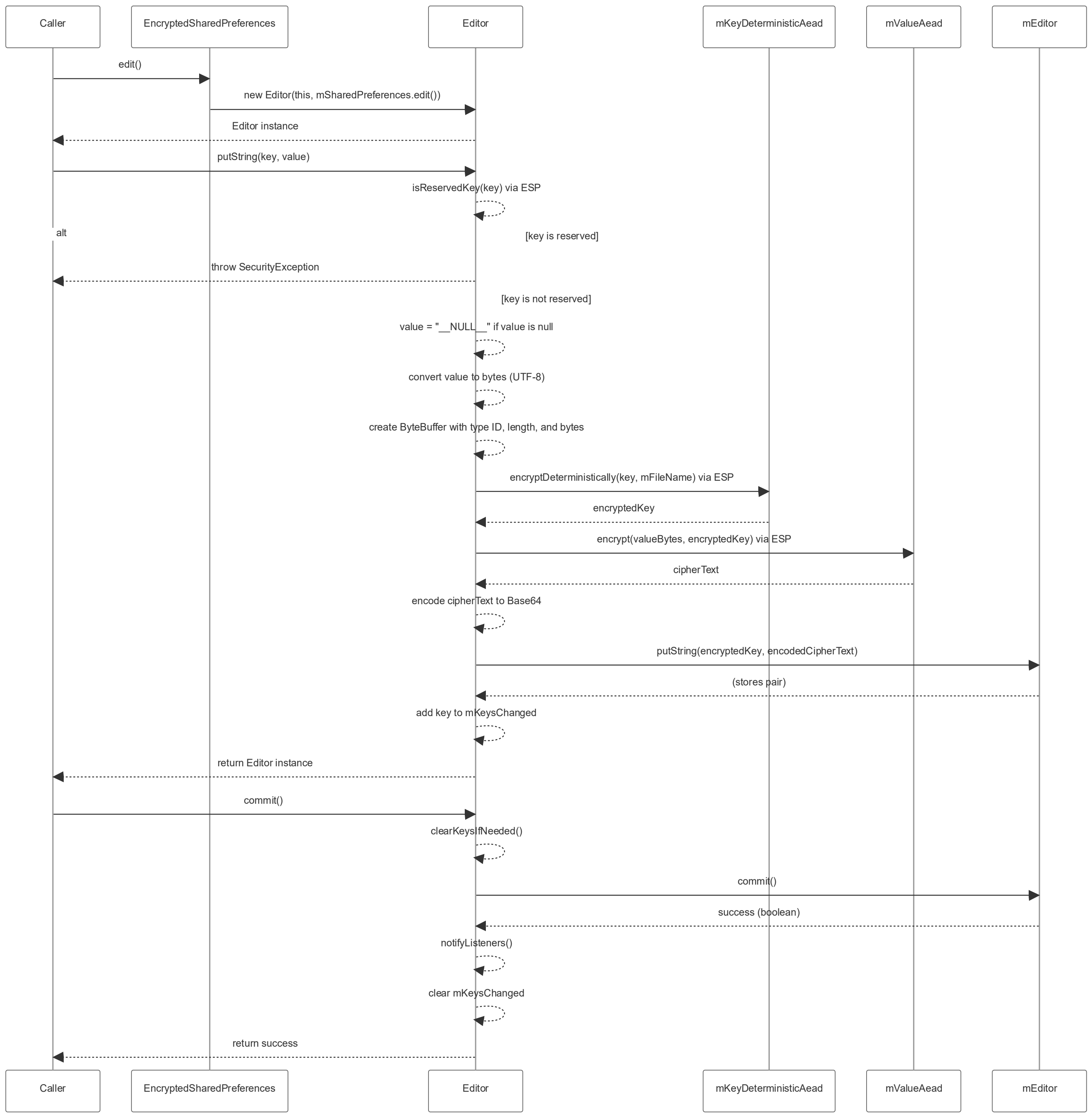

Caller
EncryptedSharedPreferences
Editor
mKeyDeterministicAead
mValueAead
mEditor
edit()
new Editor(this, mSharedPreferences.edit())
Editor instance
putString(key, value)
isReservedKey(key) via ESP
alt
[key is reserved]
throw SecurityException
[key is not reserved]
value = "__NULL__" if value is null
convert value to bytes (UTF-8)
create ByteBuffer with type ID, length, and bytes
encryptDeterministically(key, mFileName) via ESP
encryptedKey
encrypt(valueBytes, encryptedKey) via ESP
cipherText
encode cipherText to Base64
putString(encryptedKey, encodedCipherText)
(stores pair)
add key to mKeysChanged
return Editor instance
commit()
clearKeysIfNeeded()
commit()
success (boolean)
notifyListeners()
clear mKeysChanged
return success

Figure 2. Sequence diagram of writing data using EncryptedSharedPreferences

2.2.2 Reading from EncryptedSharedPreferences

To read data from files, the sequence diagram in Figure 3 is followed to retrieve the master key from keystore [7], then proceed to encrypt the “key token” passed in the get method. This token is encrypted with AES-256 SIV (which will give the same output every time), then the value is fetched from shared preferences. The encrypted value fetched is then decrypted using MasterKey with the AES-256-GCM scheme.

Once the data is decrypted, the first byte is fetched, containing the id of the type, then based on its value, the Object is returned through buffer class getters, which in its turn is tested to casting to the type provided by the EncryptedSharedPreferences getter (getString, getInt, etc.) and returns the default value if the casting is not possible.

2.3 Deprecation

The Jetpack library containing EncryptedSharedPreferences currently exists under 2 releases: Stable and Alpha.

The stable version, named 1.0.0, was last updated on the 21st of April 2021, and it brought with it stable encryption using Tink 1.5.0. The maintenance of this library was dropped by the JetSec team in early 2024 and was flagged as deprecated.

The alpha release, named 1.1.0, has been in development since June 2020, going through 6 versions, the latest being 1.1.0-alpha06 that was released in April 2023 [4]. It brought upgrades to the Tink library and other improvements in performance. The library isn’t flagged deprecated as well, yet, with no beta releases for approximately 2 years, there are no stable releases on the horizon [8].

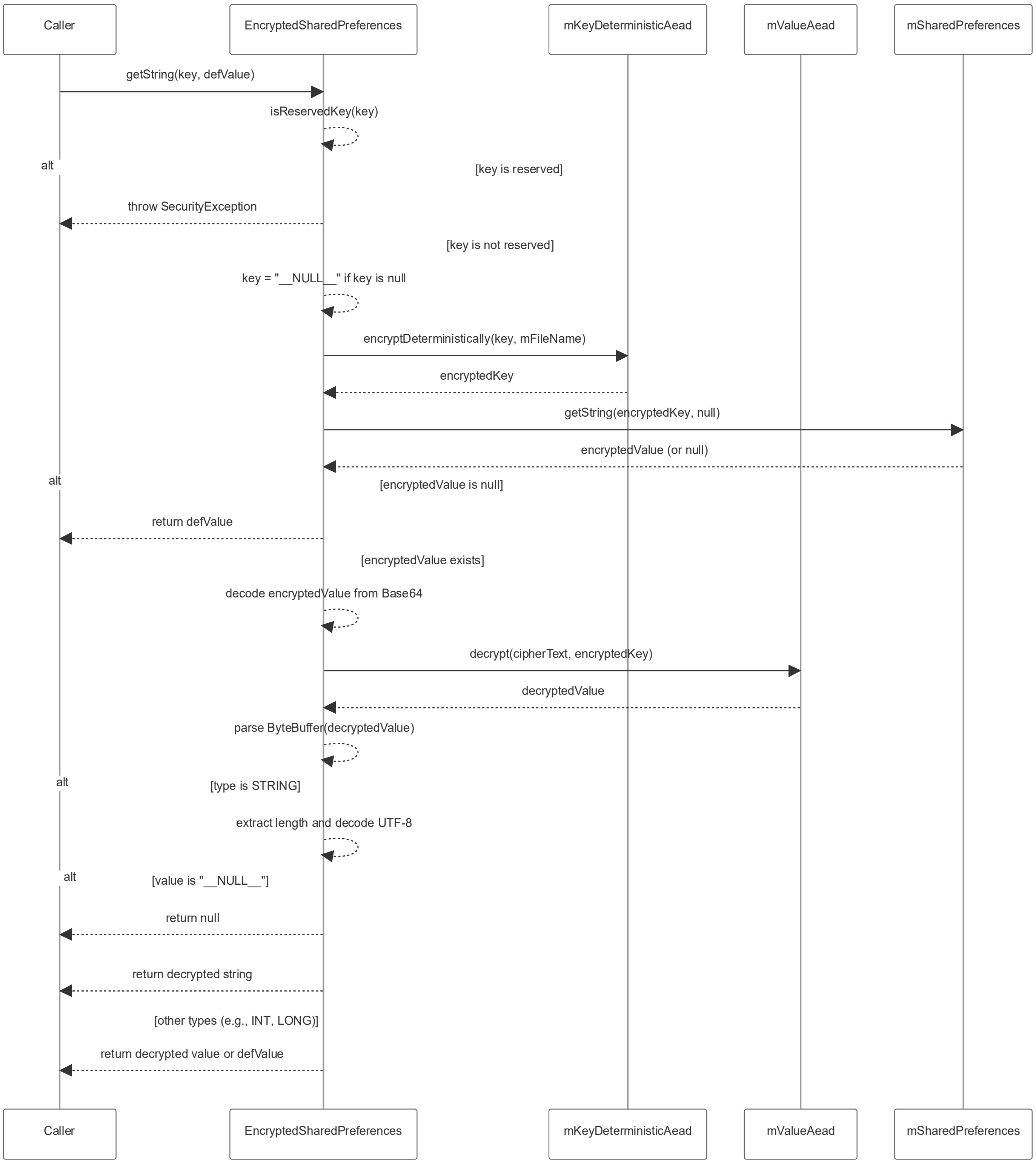


Figure 3. Sequence diagram of reading data using EncryptedSharedPreferences

3. AN ALTERNATIVE APPROACH

3.1 SharedPreferences vs. DataStore

SharedPreferences and DataStore are 2 solutions used on Android to store Key/Value pairs. SharedPreferences is the legacy method that was introduced since the first version of Android. DataStore, by contrast, was brought with Jetpack, and it is currently the recommended by Google [4].

In Table 1, a comparison following the 3S matrix (Stability, Security, and Speed) shows the differences between the SharedPreferences and DataStore.

Following this comparison, we see the leverage DataStore gives, especially when dealing with large amounts of data, complex structures, or the need to do multiple operations. In these use cases, DataStore guarantees a reduced impact on the application's performance and the integrity of the data handled.

Table 1. Comparison between SharedPreferences and DataStore [9-11]

Aspect	Feature	SharedPreferences	DataStore
Stability	Data corruption	Risk of data corruption if the application crashes during write operations	Provides transaction-based operations, guaranteeing data consistency even in crashes
	Datatypes' support	Limited support for basic data types	Support complex data types (in Proto DataStore)
	Application Not Responding (ANR) safety	Runs on the main thread, which risks causing ANRs for large data	Runs on a coroutine
Security	Data storage format	Key/Value pairs stored in clear Text on XML files	Proto DataStore saves serialized data, Preferences DataStore saves it in clear
	Encryption	No built in Encryption	No built-in encryption
Speed	Data types	Faster on simple data types	Fast on complex data types
	Impact on app speed	Significantly slows down the APP when operating on large data	Optimized for handling large datasets and parallel read/write operations

3.2 AES-256-GCM vs. XChaCha20-Poly1305

When AES-256-GCM and XChaCha20-Poly1305 are compared, we talk about two modern and secure encryption algorithms with the same purpose but different philosophies, performance, and use cases. AES-256-GCM, based on block ciphers, is supported by modern processors that provide acceleration hardware and is widely spread and battle-proven through usage in Transport Layer Security (TLS), Virtual Private Networks (VPNs), etc. [12]. This algorithm uses a "96-bit" nonce, which makes it vulnerable to nonce reutilization. On the other hand, XChaCha20 is a stream cipher [13] more optimized for software independently of hardware support, which makes it more adapted to old devices, embedded systems, or even mobiles [14, 15]. With a nonce of 192 bits, the probability of reusing the same nonce drops drastically compared to AES-256-GCM, which makes it more secure [16]. Thus, while AES-256-GCM is a solid algorithm and has the upper hand for devices with hardware acceleration, XChaCha20 is more adapted to our usage in Android devices [17].

4. PROPOSED SOLUTION

4.1 Related work

Our work is a part of a broader ecosystem of solutions for secure data storage in Android. As important as the subject is, with the fall of EncryptedSharedPreferences, many methods were shared amongst the community through blog articles, videos, forums, etc. [18]. A related approach is the "encrypted-datastore" library proposed by "Mr. Osip Fatkullin" in 2021, and which is still currently maintained [19]. In this library, "Mr. Osip" implements both a preferences datastore and a proto datastore following 2 approaches: the first using Authenticated Encryption with Associated Data (AEAD), and the second in which he migrates to using Stream AEAD. In both these ways, he performs the encryption with AES-256-GCM over the whole datastore file.

Another work is the one mentioned in the paper by Muchamad et al. [20], in which he implements the Advanced Encryption Standard – Cipher Block Chaining (AES-CBC) algorithm to encrypt the datastore. Similarly, Klymenko et al. [21] proposed an AES-CBC-PKCS7 encryption scheme integrated into Proto DataStore via a custom serializer, targeting secure storage of authentication tokens. Their method encrypts serialized objects (e.g., JavaScript Object Notation (JSON)-encoded user preferences) as a single blob, leveraging Android KeyStore for key generation and emphasizing protection against malware-induced data leaks.

The following Table 2 shows a comparison between Klymenko et al.'s [21] adopted approach and this paper's suggested alternative, highlighting the differences in threat models, metadata exposure, key management, and performance trade-offs.

Table 2. Comparison between the Advanced Encryption Standard – Cipher Block Chaining (AES-CBC) DataStore serializer and the proposed approach [21]

Dimension	AES-CBC DataStore Serializer	EncryptedDataStore (AES-SIV + XChaCha20-Poly1305)
Threat model	Protects against offline filesystem attackers (ADB backup, rooted FS access). Assumes trusted runtime, no active tampering, no chosen-ciphertext attacks. Confidentiality-only.	Protects against offline attackers + structural storage analysis. Assumes attacker can observe and manipulate stored data. Designed for active tampering resistance and authenticated reads.
Metadata exposure	Leaks file size, update timing, and logical structure (entire DataStore encrypted as one unit; no key-level hiding). Preference keys are implicitly exposed at the serialization boundary.	Leaks only file size and timing. Preference keys are encrypted deterministically (AES-SIV), hiding semantic meaning and access patterns. No plaintext metadata inside Preferences.
Key management	Single symmetric AES key stored in Android Keystore. No key separation: compromise exposes the entire store. No key hierarchy or derivation.	Hierarchical key model: master key in Keystore (via Tink) → derived per-key/value encryption keys. Strong domain separation (keys vs. values). Reduced blast radius on compromise.
Performance trade-offs	Low Central Processing Unit (CPU) cost, but high Input/Output (I/O) amplification: every write re-encrypts the entire blob; every read decrypt full payload. Scales poorly beyond small configs.	Moderate CPU cost (AEAD + derivation), but minimal I/O: encrypt/decrypt only accessed entries. Scales well with large or frequently updated preference sets.

4.2 Algorithms and diagrams

4.2.1 The premise

From a black box point of view, the implementation is intended to follow a logic that respects the two main conditions mentioned below in Figures 4 and 5. While the library "Preferences DataStore" provided by Google gives the ability to store and retrieve data, it also provides us with the base block of storage management.

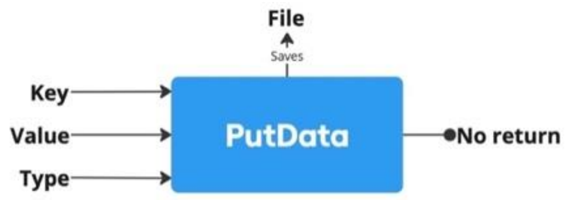


Figure 4. Save data function block diagram

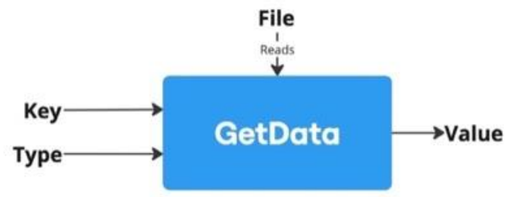


Figure 5. Get data function block diagram

4.2.2 The challenges

Our goal is to make that library easy to adopt with a minimal migration cost, fast operations, and data integrity. This puts us in front of multiple possible implementations.

• Add extensions to DataStore: This solution consists of creating two new methods, "getEncrypted" and "setEncrypted" that rely on the default getter and setter but with a layer managing encryption/decryption. This solution is the fastest in terms of cost of dev, but it creates

• Make a DataStore delegate and factory that uses EncryptedFile: As used by "Mr. Osip Fatkullin" in his proposed solution, yet this method has two flaws. The first is that EncryptedFile is in the Jetpack security-crypto library, which was deprecated and contained EncryptedSharedPreferences as well. The second flaw is in the read and write operations, having a file encryption forces the whole file to be reencrypted with every writing operation, which creates delays and might create concurrency issues with multiple read and write operations.

• Override the getters and setters of the Preferences class used originally by DataStore: Although this solution sounds appealing and perfect, it is not possible. These methods are defined as final with internal getters, setters, and constructors. So simply, they can't be inherited and can't be overridden.

4.2.3 The solution

These challenges were addressed by designing a modular architecture composed of multiple interacting components that collectively implement a seamless encryption layer atop the existing DataStore (see Figure 6). These components are as follows:

EncryptedDataStore: The core module that extends the standard DataStore functionality, adding an editEncrypted method that sets the scope of encryption, encryptedData returning a flow of encryptedData and encryptionOptions that hold the key, value encryption strategies, etc. This module mainly associates encryptionOptions with instances of DataStore<Preferences>.

EncryptedPreferenceKeys: It is a dynamic key-generation utility that extends the ability to store only primary data types, sets, and byte arrays to store any type of data. It leverages reflection to create these keys at runtime.

EncryptedPreferencesDataStoreDelegate: A delegate-based implementation that ensures lazy initialization and secure management of the encryptedDataStore instance. It takes an optional EncryptionOptions, which it passes to DataStore.encryptionOptions.

EncryptedPreferencesSerializer: A serialization module that encodes and decodes various data types. It supports primitive types, string sets, byte arrays, and complex data types that are annotated @Serializable.

EncryptionOptions: It represents the configuration class that aggregates the encryption strategies and serialization settings, allowing customization of key and value customization approaches.

EncryptionStrategy: An interface defining two methods, encrypt and decrypt, that is used in the default KeyEncryptionStrategy and ValueEncryptionStrategy.

KeyEncryptionStrategy: Implements AES-256-SIV for encrypting preference keys, ensuring metadata confidentiality

ValueEncryptionStrategy: Implements XChaCha20-Poly1305 to encrypt preference values, providing fast authenticated encryption.

As shown in Figure 7, the first step was designing a delegate for EncryptedDataStore, which serves as the entry point to instantiate the data store using the library's default factory.

Since the standard Preferences and MutablePreferences classes cannot be modified, we introduced an EncryptedMutablePreferences class equipped with dynamic get and set operators. These methods leverage the underlying operators of MutablePreferences, ensuring a seamless experience for developers. This class is managed within the data store through an extended method called editEncrypted.

For security, EncryptedDataStoreHelper was made to handle the encryption and decryption of both keys and values before storing them in preferences. The EncryptionOptions class was added to encapsulate the KeyEncryptionStrategy for keys, the ValueEncryptionStrategy for values, and a serializer that processes data before storage. Encryption operations and key management are powered by Google's "Tink" library, which securely stores master keys in the Keystore and key-value-specific keys in shared preferences.

When it comes to saving data (see Figure 8), we implemented a streamlined workflow that is initiated when invoking editEncrypted on a DataStore<Preferences> instance, which you'd typically set up using our encryptedPreferencesDataStore function. You pass in a Preferences.Key<T> and the plain-text value inside a transform block. Inside that block, EncryptedMutablePreferences is instantiated with the mutable preferences and the EncryptionOptions we've configured. The set method here triggers "EncryptedDataStoreHelper.setEncrypted", where the encryption process is executed: KeyEncryptionStrategy is used to encrypt the key with AES-256-SIV and ValueEncryptionStrategy to encrypt the value with XChaCha20-Poly1305. Before encryption, our EncryptedPreferencesSerializer steps in to serialize the value into a string—think Base64 for byte arrays or JSON for custom objects. Once encrypted, the data gets stored in the underlying MutablePreferences, and the whole transaction wraps up atomically via updateData, ensuring everything's persisted securely in DataStore.

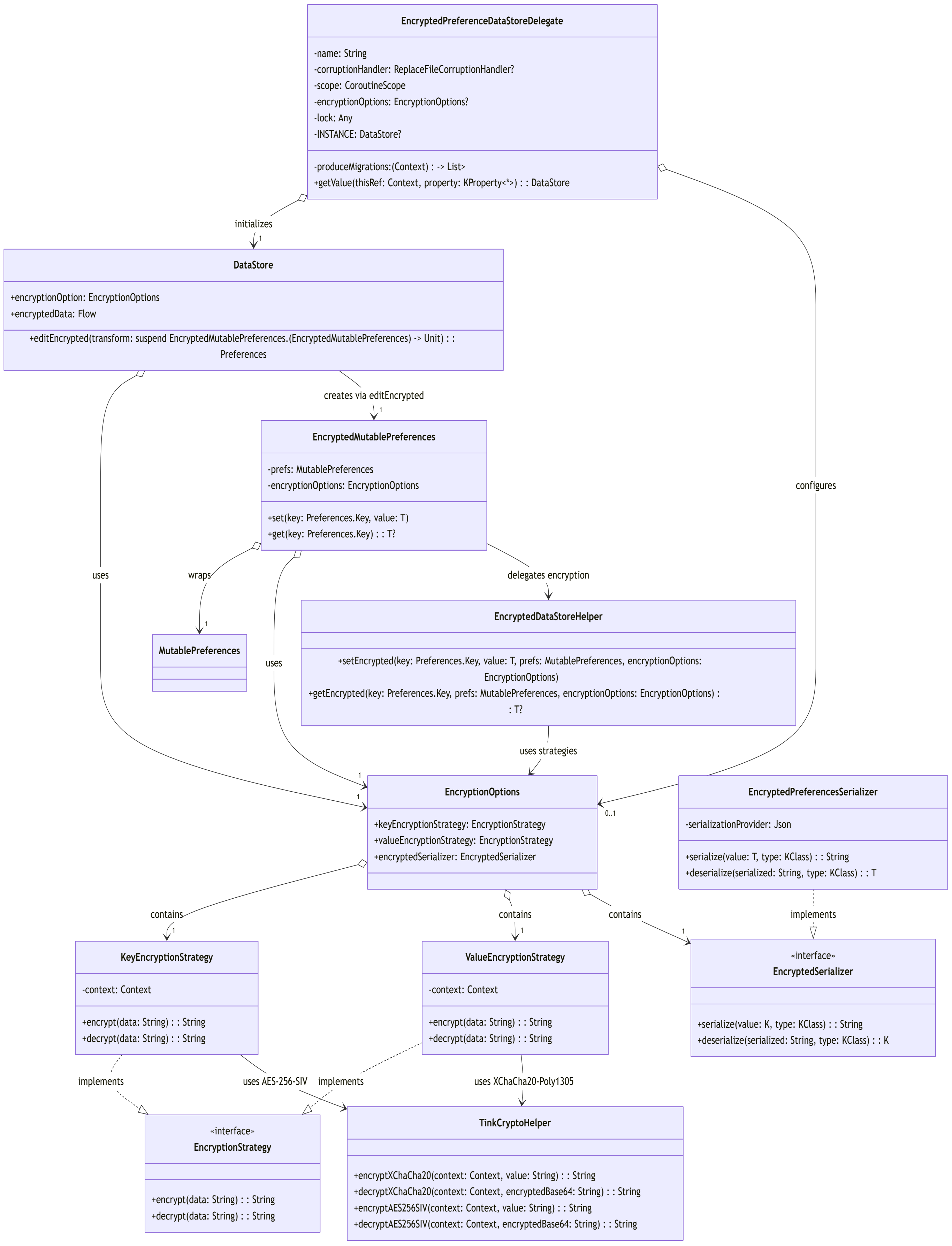

EncryptedPreferenceDataStoreDelegate
-name: String
-corruptionHandler: ReplaceFileCorruptionHandler?
-scope: CoroutineScope
-encryptionOptions: EncryptionOptions?
-lock: Any
-INSTANCE: DataStore?
-produceMigrations:(Context) : -> List>
+getValue(thisRef: Context, property: KProperty<*>) : : DataStore
initializes
1
configures
DataStore
+encryptionOption: EncryptionOptions
+encryptedData: Flow
+editEncrypted(transform: suspend EncryptedMutablePreferences.(EncryptedMutablePreferences) -> Unit) : : Preferences
creates via editEncrypted
1
EncryptedMutablePreferences
-prefs: MutablePreferences
-encryptionOptions: EncryptionOptions
+set(key: Preferences.Key, value: T)
+get(key: Preferences.Key) : : T?
uses
wraps
delegates encryption
1
MutablePreferences
uses
EncryptedDataStoreHelper
+setEncrypted(key: Preferences.Key, value: T, prefs: MutablePreferences, encryptionOptions: EncryptionOptions)
+getEncrypted(key: Preferences.Key, prefs: MutablePreferences, encryptionOptions: EncryptionOptions) : : T?
uses strategies
1
1
EncryptionOptions
+keyEncryptionStrategy: EncryptionStrategy
+valueEncryptionStrategy: EncryptionStrategy
+encryptedSerializer: EncryptedSerializer
0..1
EncryptedPreferencesSerializer
-serializationProvider: Json
+serialize(value: T, type: KClass) : : String
+deserialize(serialized: String, type: KClass) : : T
contains
contains
contains
implements
1
1
1
KeyEncryptionStrategy
-context: Context
+encrypt(data: String) : : String
+decrypt(data: String) : : String
ValueEncryptionStrategy
-context: Context
+encrypt(data: String) : : String
+decrypt(data: String) : : String
«interface»
EncryptedSerializer
+serialize(value: K, type: KClass) : : String
+deserialize(serialized: String, type: KClass) : : K
implements
uses AES-256-SIV
implements
uses XChaCha20-Poly1305
«interface»
EncryptionStrategy
+encrypt(data: String) : : String
+decrypt(data: String) : : String
TinkCryptoHelper
+encryptXChaCha20(context: Context, value: String) : : String
+decryptXChaCha20(context: Context, encryptedBase64: String) : : String
+encryptAES256SIV(context: Context, value: String) : : String
+decryptAES256SIV(context: Context, encryptedBase64: String) : : String

Figure 6. Class diagram of EncryptedDataStore library

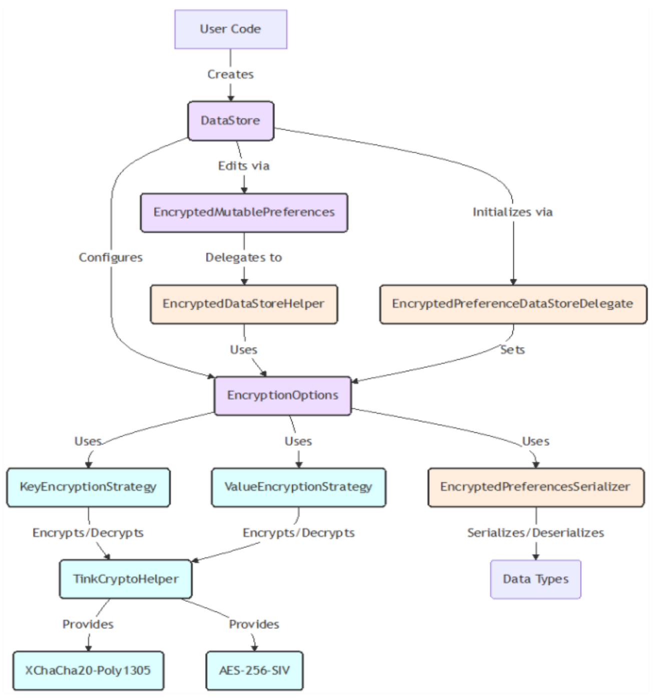


Figure 7. EncryptedDataStore library graph

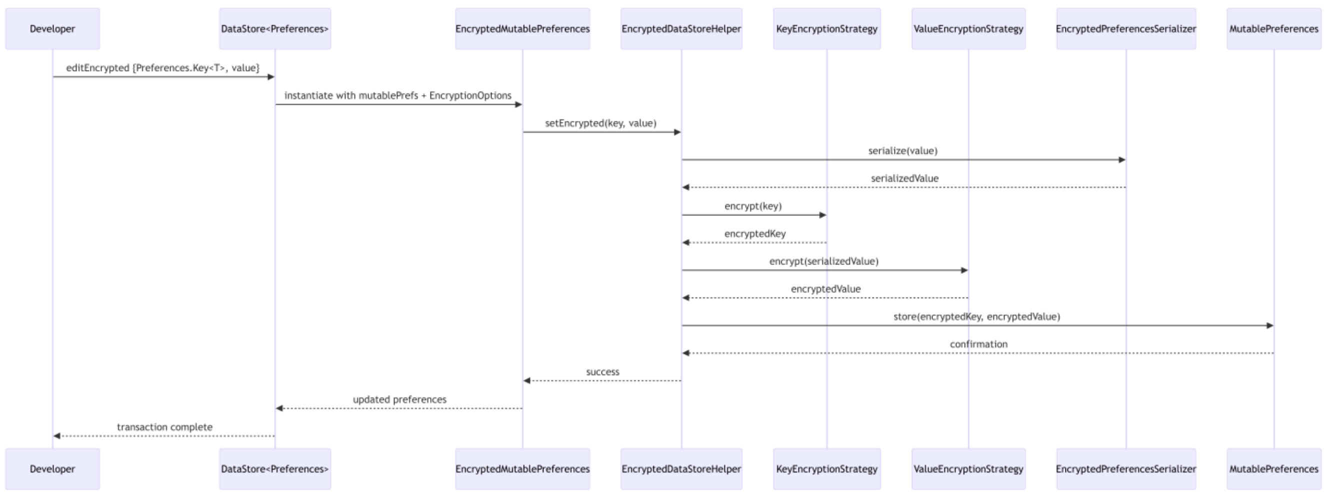


Figure 8. Sequence diagram for writing data using EncryptedDataStore

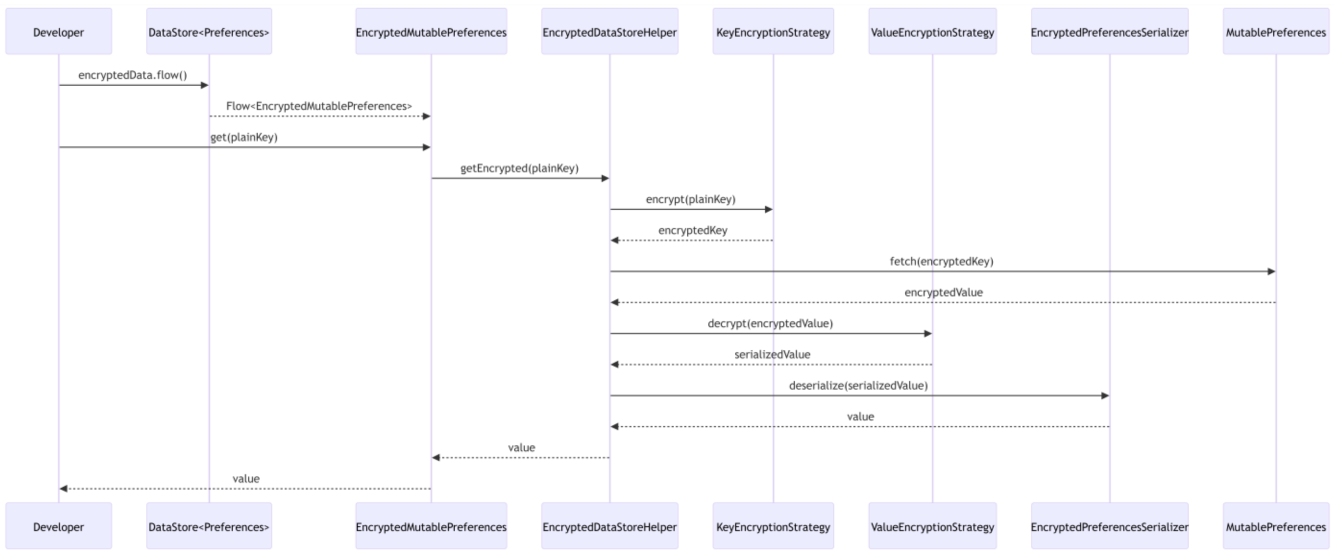


Figure 9. Sequence diagram for reading data using EncryptedDataStore

For reading data, we designed it to be just as seamless. You start by tapping into the encryptedData property on DataStore<Preferences>, which gives a Flow<EncryptedMutablePreferences>, as seen in Figure 9. From there, call get on EncryptedMutablePreferences with the plain-text key. This initiates EncryptedDataStoreHelper.getEncrypted, where that key is encrypted with AES-256-SIV using KeyEncryptionStrategy to match the encrypted key in storage. Once the encrypted value is located, ValueEncryptionStrategy decrypts it with XChaCha20-Poly1305. Then, EncryptedPreferencesSerializer deserializes the decrypted string back to its original type—reversing the Base64 or JSON process as needed. The value is then returned, secure and ready to use.

5. SECURITY CHECKS

To assess the library, a set of tests is run over the demo app. The tests will be over two levels: "Data at rest" and "Data in processing".

5.1 Tools and environment

• Attacker Machine: MacBook pro M4 Pro on macOS Sequoia 15.3.2
• Target Device: Pixel 8a AVD on Android 15
• ADB version: 1.0.41
• Platform-Tools version: 35.0.2
• Jadx version: 1.5.1
• Frida version: 16.6.6
• RootAVD and Magisk

Rooting the AVD was inspired by the post written in Medium by a security researcher and Bug bounty hunter known as Snaggy [22].

5.2 Data at rest

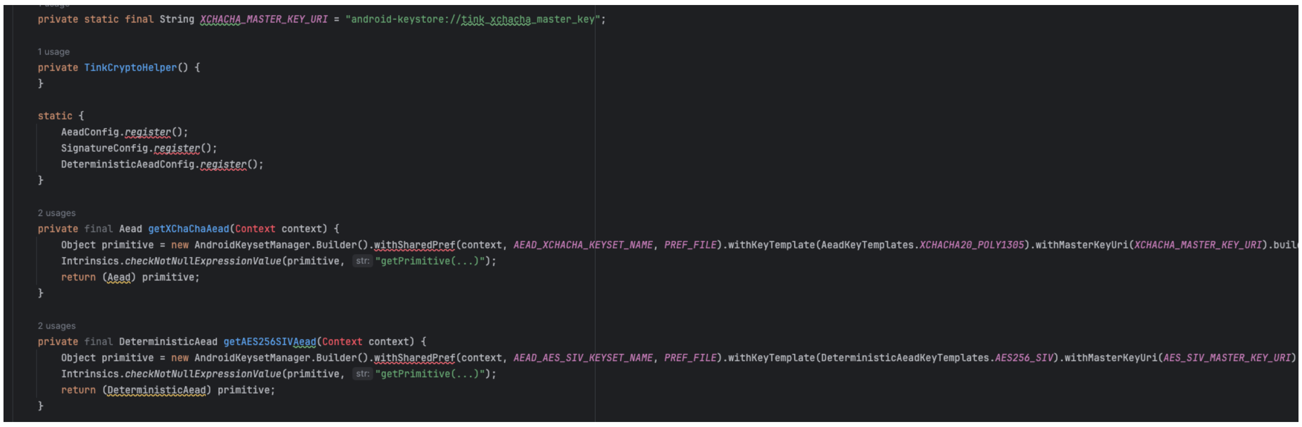
Objective: Assess whether encrypted data stored by the library is protected from unauthorized extraction and whether sensitive metadata or keys are exposed.

5.2.1 Static analysis

• Method: The Android Package Kit (APK) was decompiled using jadx -d output/ app.apk and files opened in Android Studio (see Figure 10).
• Test Criteria:
-Pass: No hardcoded keys or secrets; library-level cryptographic primitives not trivially exposed.
-Fail: Hardcoded keys, exposed algorithm URIs, or easily recoverable cryptographic parameters.
-Findings (see Table 3).
• Recommendation: Add obfuscation to the library using R8.

5.2.2 Filesystem security

• Method: Attempted to access data files via Android Debug Bridge (ADB) shell and Android file explorer (see Figures 11-13).
• Test Criteria:
-Pass: No hardcoded keys or secrets; library-level cryptographic primitives not trivially exposed.
-Fail: Hardcoded keys, exposed algorithm URIs, or easily recoverable cryptographic parameters.
-Findings (see Table 4).
• Conclusion: Library properly encrypts files at rest, but root access circumvents OS-level protections, as expected.


```
private static final String XCHACHA_MASTER_KEY_URI = "android-keystore://tink_xchacha_master_key";

1 usage
private TinkCryptoHelper() {
}

static {
    AeadConfig.register();
    SignatureConfig.register();
    DeterministicAeadConfig.register();
}

2 usages
private final Aead getXChaChaAead(Context context) {
    Object primitive = new AndroidKeysetManager.Builder().withSharedPref(context, AEAD_XCHACHA_KEYSET_NAME, PREF_FILE).withKeyTemplate(AeadKeyTemplates.XCHACHA20_POLY1305).withMasterKeyUri(XCHACHA_MASTER_KEY_URI).buil
    Intrinsics.checkNotNullExpressionValue(primitive, str: "getPrimitive(...)");
    return (Aead) primitive;
}

2 usages
private final DeterministicAead getAES256SIVAead(Context context) {
    Object primitive = new AndroidKeysetManager.Builder().withSharedPref(context, AEAD_AES_SIV_KEYSET_NAME, PREF_FILE).withKeyTemplate(DeterministicAeadKeyTemplates.AES256_SIV).withMasterKeyUri(AES_SIV_MASTER_KEY_URI)
    Intrinsics.checkNotNullExpressionValue(primitive, str: "getPrimitive(...)");
    return (DeterministicAead) primitive;
}
```

Figure 10. TinkCryptoHelper.java in the jadx output

Table 3. Results of Static analysis of the decompiled Android Package Kit (APK)

Aspect	Observation	Level	Evaluation
Encryption keys	Not Hardcoded	Library	Pass
Master key storage	Managed in Android KeyStore	Library	Pass
Cryptographic metadata	TinkCryptoHelper exposes algorithm identifiers and Universal Resource Identifiers (URIs)	Library	Warning
Code obfuscation	None	Application and Library	Warning

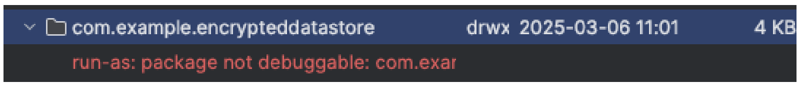
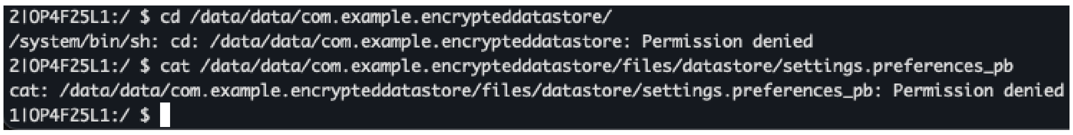
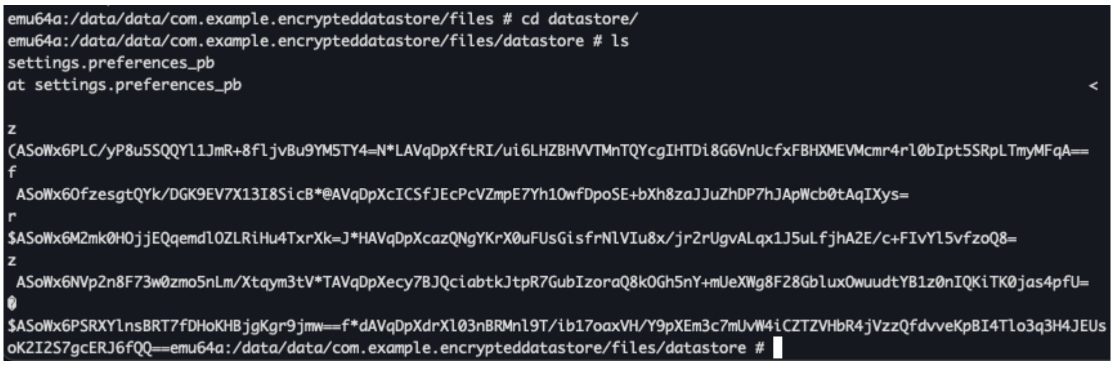

```
⌄ com.example.encrypteddatastore          drwx  2025-03-06 11:01      4 KB
    run-as: package not debuggable: com.exar
```

Figure 11. Results trial to access via Android file explorer



```
2|OP4F25L1:/ $ cd /data/data/com.example.encrypteddatastore/
/system/bin/sh: cd: /data/data/com.example.encrypteddatastore: Permission denied
2|OP4F25L1:/ $ cat /data/data/com.example.encrypteddatastore/files/datastore/settings.preferences_pb
cat: /data/data/com.example.encrypteddatastore/files/datastore/settings.preferences_pb: Permission denied
1|OP4F25L1:/ $
```

Figure 12. Results of shell commands execution through Android Debug Bridge (ADB) shell



```
emu64a:/data/data/com.example.encrypteddatastore/files # cd datastore/
emu64a:/data/data/com.example.encrypteddatastore/files/datastore # ls
settings.preferences_pb
at settings.preferences_pb                                                                      <

z
(ASoWx6PLC/yP8u5SQQYl1JmR+8fljvBu9YM5TY4=N*LAVqDpXftRI/ui6LHZBHVVTMnTQYcgIHTDi8G6VnUcfxFBHXMEVMcmr4rl0bIpt5SRpLTmyMFqA==
f
 ASoWx6OfzesgtQYk/DGK9EV7X13I8SicB*@AVqDpXcICSfJEcPcVZmpE7Yh1OwfDpoSE+bXh8zaJJuZhDP7hJApWcb0tAqIXys=
r
$ASoWx6M2mk0HOjjEQqemdlOZLRiHu4TxrXk=J*HAVqDpXcazQNgYKrX0uFUsGisfrNlVIu8x/jr2rUgvALqx1J5uLfjhA2E/c+FIvYl5vfzoQ8=
z
 ASoWx6NVp2n8F73w0zmo5nLm/Xtqym3tV*TAVqDpXecy7BJQciabtkJtpR7GubIzoraQ8kOGh5nY+mUeXWg8F28GbluxOwuudtYB1z0nIQKiTK0jas4pfU=
@
$ASoWx6PSRXYlnsBRT7fDHoKHBjgKgr9jmw==f*dAVqDpXdrXl03nBRMnl9T/ib17oaxVH/Y9pXEm3c7mUvW4iCZTZVHbR4jVzzQfdvveKpBI4Tlo3q3H4JEUs
oK2I2S7gcERJ6fQQ==emu64a:/data/data/com.example.encrypteddatastore/files/datastore #
```

Figure 13. Result of shell commands through Android Debug Bridge (ADB) on rooted device

Table 4. Results of attempts to access data files via Android Debug Bridge (ADB) and file explorer

Aspect	Observation	Level	Evaluation
File accessibility (non-root)	Not accessible	Operating System (OS) and Library	Pass
File accessibility (root)	Accessible	OS and Library	Warning
Data confidentiality	Stored data and metadata encrypted	Library	Pass

5.3 Data in processing

Objective: Evaluate potential data leakage while the application is running.

5.3.1 Memory dump

• Method: Used Frida to capture memory of the running process. (see Figures 14 and 15).

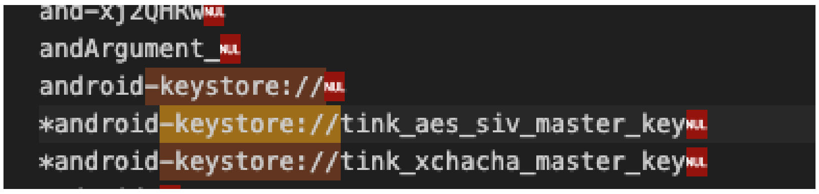
• Test Criteria:

-Pass: Only encrypted data is observable; no intermediate cryptographic keys or plaintext values remain in memory beyond a minimal scope.

-Fail: Keys or unencrypted values can be reconstructed from memory.

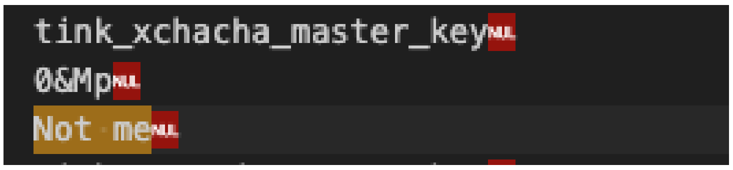
• Findings (see Table 5).

• Conclusion: The library maintains encryption for data in processing. Observed plaintext exposures are primarily due to the behavior of demo applications, not library deficiencies. Developers should avoid storing sensitive data in unencrypted application variables.



```
and-xj2QHRW
andArgument_
android-keystore://
*android-keystore://tink_aes_siv_master_key
*android-keystore://tink_xchacha_master_key
```

Figure 14. A fragment of Frida's dump Strings.txt file showing keystore paths



```
tink_xchacha_master_key
0&Mp
Not me
```

Figure 15. A fragment of Frida's dump Strings.txt file showing unencrypted data that was in memory when it was dumped

Table 5. Results of memory dump using Frida

Aspect	Observation	Level	Evaluation
Data in memory	Only encrypted values are visible from the library	Library	Pass
Key paths	Keystore paths visible	Library	Warning
Application variables	Plaintext data visible (demo app)	Application	Warning
Intermediate key exposure	Potentially accessible during memory dump if Tink relies on garbage collection	Library	Warning

6. PERFORMANCE REVIEW AND COMPARISON

To evaluate the performance of the EncryptedDataStore library, we conducted microbenchmarks comparing it against the deprecated EncryptedSharedPreferences (using the stable 1.0.0 version) and the unencrypted Preferences DataStore. These benchmarks focused on write and read operations for small data payloads (e.g., simple key-value pairs like strings or integers, in our case, it's the word "Hello") and larger payloads (50 KB serialized data blocks), executed over 50 sequential runs on an Oppo Reno5 running colorOS 13.0 based on Android 13. The metrics captured execution time in nanoseconds (ns), highlighting latency, variability, and overhead introduced by encryption. All tests were performed in a controlled environment to minimize external interference, with results visualized in Figures 16-19.

6.1 Analysis of write operations

For small write operations (see Figure 16), the EncryptedDataStore exhibits higher latency, ranging from approximately 0.6×10^8 ns to 1.6×10^8 ns (60–160 ms), with notable fluctuations across runs. This variability suggests sensitivity to factors such as key generation, serialization, and per-entry encryption overhead using AES-256-SIV for keys and XChaCha20-Poly1305 for values. In contrast, EncryptedSharedPreferences (orange line) and Preferences DataStore (green line) maintain near-negligible times, hovering around $0–0.2 \times 10^8$ ns, with minimal variance. The overhead in EncryptedDataStore is attributable to its granular encryption approach, which processes each key-value pair individually, unlike EncryptedSharedPreferences's file-level encryption or the unencrypted baseline.

For larger writes (see Figure 17), the pattern persists but with amplified effects: EncryptedDataStore latencies fluctuate between 6×10^7 ns and 9×10^7 ns (60–90 ms), while EncryptedSharedPreferences shows spikes up to 3×10^7 ns in isolated runs but generally remains lower ($0–2 \times 10^7$ ns average), and Preferences DataStore is consistently flat near zero. The increased overhead for EncryptedDataStore on larger payloads underscores the computational cost of stream-cipher encryption (XChaCha20-Poly1305) and serialization, particularly when handling byte arrays or complex objects, which involve additional Base64 or JSON processing.

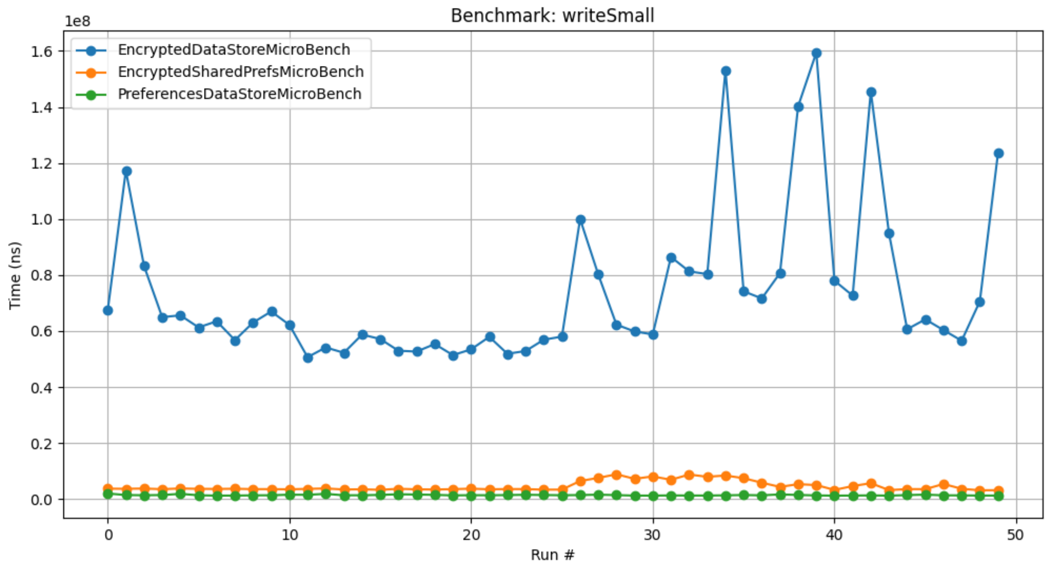


Figure 16. Benchmark of writing 5 bytes using the three libraries

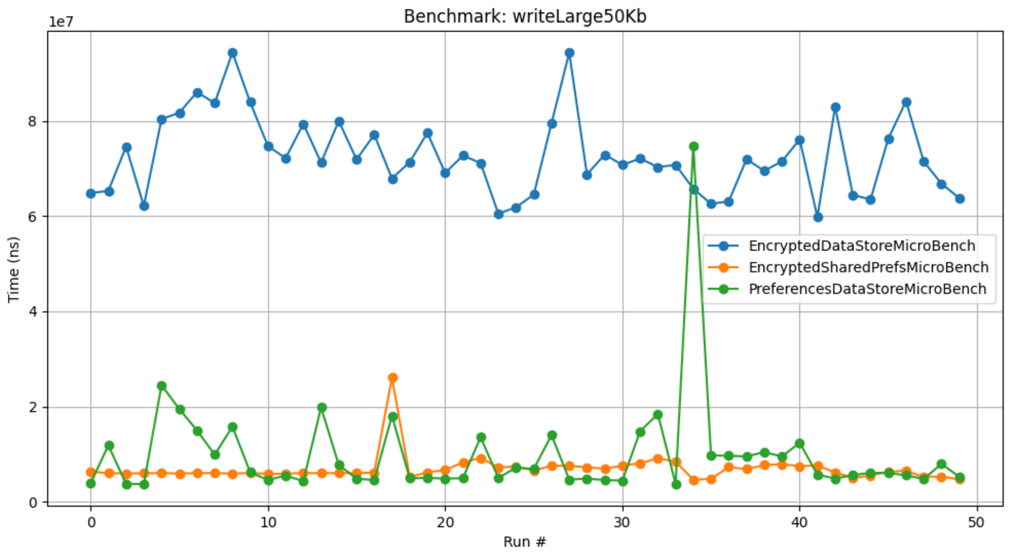


Figure 17. Benchmark of writing a 50 KB stream using the 3 libraries

6.2 Analysis of read operations

Read operations mirror the write trends. In small reads (see Figure 18), EncryptedDataStore times range from 5×10^7 ns to 7×10^7 ns (50–70 ms), again with high fluctuation, compared to the stable, low latencies of the baselines (near 0 ns). This indicates decryption and deserialization as primary bottlenecks, exacerbated by the need to encrypt the query key for lookup and then decrypt the value.

For large reads (see Figure 19), latencies for EncryptedDataStore climb to 6×10^7 ns – 9×10^7 ns, like large writes, while competitors remain orders of magnitude faster (even with Datastore being 4 times slower than base encrypted shared preferences [23]). The consistency in overhead across read/write and payload sizes points to inherent costs in the Tink library's key management and the per-entry encryption model, which, while enhancing security through individualized protection, introduces measurable delays.

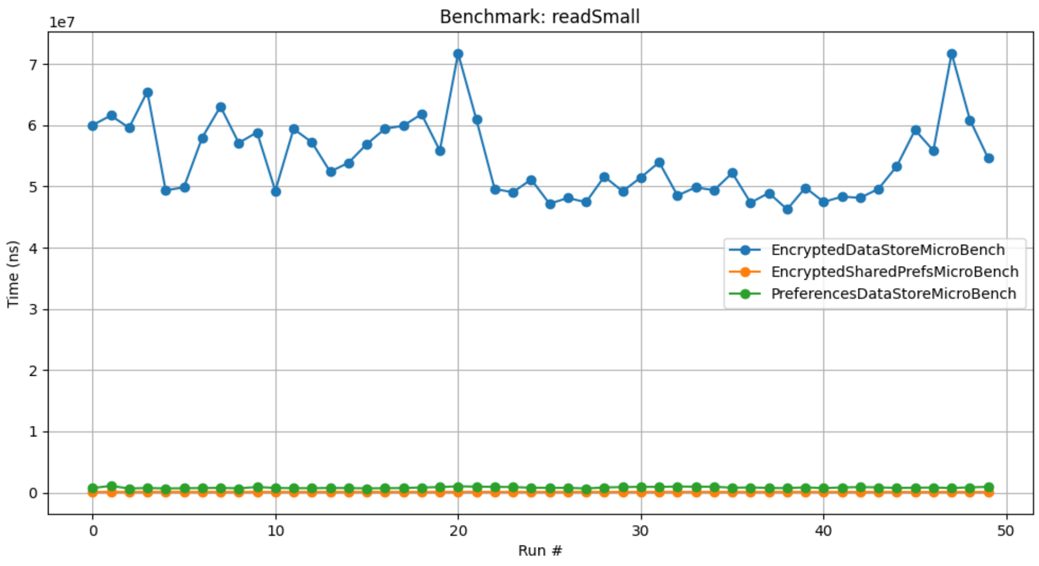


Figure 18. Benchmark of reading 5 bytes using the 3 libraries

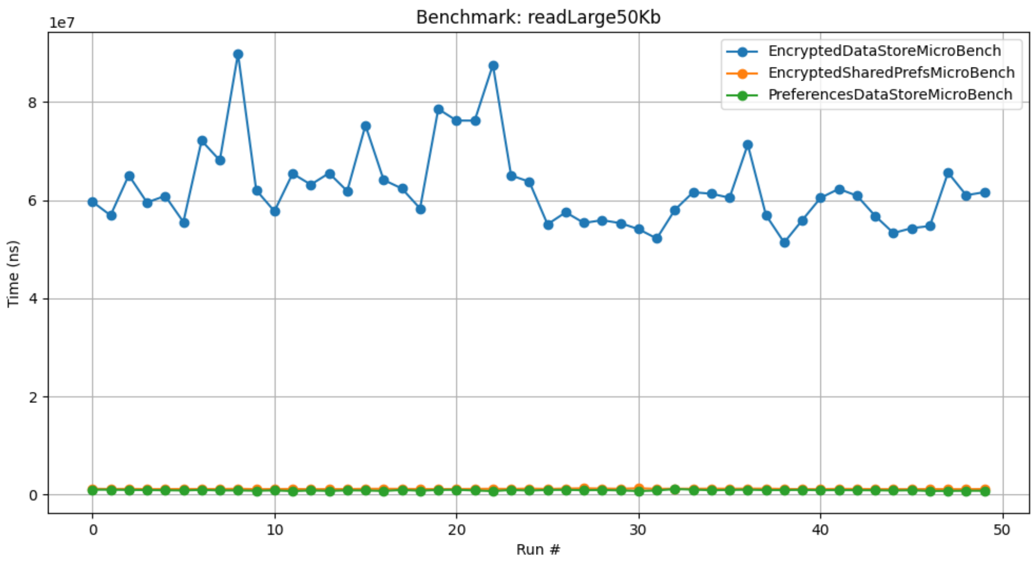


Figure 19. Benchmark of reading a 50 KB stream using the 3 libraries

6.3 Comparative insights and ethical considerations

Frankly, these results reveal that EncryptedDataStore underperforms relative to both EncryptedSharedPreferences and the unencrypted Preferences DataStore, with overhead factors of 10–100× in latency. This is not unexpected, as our design prioritizes security (e.g., metadata confidentiality via key encryption and resistance to nonce reuse via XChaCha20's larger nonce) over raw speed, aligning with Google's shift toward DataStore for better data integrity and asynchronous operations. However, the elevated and variable times highlight inefficiencies, such as repeated key derivations or serialization steps, which could impact user experience in high-frequency access scenarios. Academically, this underscores a trade-off between security and performance in mobile storage systems, a common theme in the literature where encryption inevitably adds computational burden, especially on resource-constrained devices.

The slower performance of EncryptedDataStore, while a limitation, provides substantial space for improvement. Potential optimizations include leveraging hardware-accelerated cryptography (e.g., via Android's Hardwarebacked Keystore), batching operations to amortize encryption costs, or hybrid approaches that cache decrypted values in memory for short-lived sessions (with careful invalidation to maintain security). Future iterations could also explore lighter serialization formats or profile Tink's internals for bottlenecks. The possibility of changing the encryption algorithm to XTS-AES is impractical in this context since it’s more adapted to disk/partition encryption and not key-value use cases.

7. CONCLUSION AND FUTURE WORK

This study addressed the challenge introduced by the 2024 deprecation of EncryptedSharedPreferences by proposing EncryptedDataStore, a secure storage mechanism built atop Android’s Preferences DataStore. The proposed design integrates AES-256-SIV for deterministic key encryption and XChaCha20-Poly1305 for authenticated value encryption, relying on the Android Keystore and Google’s Tink library for cryptographic key management. Through a series of security evaluations—including static analysis, filesystem inspection, and runtime memory analysis on rooted devices—the solution demonstrated effective protection of sensitive data at rest and resistance to straightforward data extraction attempts.

The performance evaluation reveals a pronounced trade-off between security and efficiency. Experimental results show that EncryptedDataStore incurs a significant latency overhead, ranging from one to two orders of magnitude compared to both unencrypted Preferences DataStore and EncryptedSharedPreferences. This overhead is primarily attributable to per-entry encryption, serialization and deserialization costs, and cryptographic key handling. Consequently, while the proposed solution strengthens confidentiality and integrity guarantees and aligns with Android’s recommended DataStore architecture, it does not offer performance improvements over existing mechanisms. Its suitability is therefore limited to scenarios where security requirements outweigh strict latency constraints.

Future work will focus on addressing the limitations identified in this evaluation. First, the encryption model will be extended to Proto DataStore, where structured data and transactional semantics may enable more efficient encryption strategies. Second, a comprehensive security audit will be conducted to assess resilience against advanced attack vectors, including side-channel and runtime manipulation attacks. Finally, optimization efforts will investigate reducing cryptographic overhead through improved serialization mechanisms, selective caching strategies, and more efficient key usage, to improve performance while preserving the security properties demonstrated in this work.

REFERENCES

[1] Duarte, F. (2025). Amount of data created daily. Exploding Topics. https://explodingtopics.com/blog/data-generated-per-day.

[2] Holloway-George, E. (2024). Securing the future: Navigating the deprecation of encrypted shared preferences. ProAndroidDev. https://proandroiddev.com/securing-the-future-navigating-the-deprecation-of-encrypted-shared-preferences-91ce3c20ae8d.

[3] Garin, F. (2012). Android (3rd edition). Dunod.

[4] EncryptedSharedPreferences. https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences.

[5] Security-crypto. https://android.googlesource.com/platform/frameworks/support/+/androidx-main/security/security-crypto.

[6] Harkins, D. (2008). Synthetic Initialization Vector (SIV) authenticated encryption using the Advanced Encryption Standard (AES). https://datatracker.ietf.org/doc/html/rfc5297.

[7] Cooijmans, T., de Ruiter, J., Poll, E. (2014). Analysis of secure key storage solutions on Android. In Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, Scottsdale, Arizona, USA, pp. 11-20. https://doi.org/10.1145/2666620.2666627

[8] Jetpack. https://developer.android.com/jetpack/androidx/releases/security.

[9] Kornaś, D. (2021). Analysis of data storage methods available in the Android SDK. Journal of Computer Sciences Institute, 21: 378-382. https://doi.org/10.35784/jcsi.2759

[10] Casolare, R., Martinelli, F., Mercaldo, F., Santone, A. (2020). Android collusion: Detecting malicious applications inter-communication through SharedPreferences. Information, 11(6): 304. https://doi.org/10.3390/info11060304

[11] Hagos, T. (2018). Learn Android Studio 3 with Kotlin. Learn Android Studio 3 with Kotlin: Efficient Android App Development. Berkeley, CA: Apress.

[12] Thirupalu, U., Reddy, E.K. (2020). Performance analysis of cryptographic algorithms in the information security. International Journal of Engineering Research and Technology, 8(2): 64-69. https://doi.org/10.13140/RG.2.2.16273.51047

[13] De Santis, F., Schauer, A., Sigl, G. (2017). ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications. In Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, pp. 692-697. https://doi.org/10.23919/DATE.2017.7927078

[14] Kao, T.L., Wang, H.C., Li, J.E. (2021). Safe MQTT-SN: A lightweight secure encrypted communication in IoT. Journal of Physics: Conference Series, 2020(1): 012044. https://doi.org/10.1088/1742-6596/2020/1/012044

[15] Dzahabi, Z.Y., Hayaty, N., Bettiza, M. (2025). Cryptography of Chacha20 and RSA algorithms for text security. Journal of Computer Networks, Architecture and High Performance Computing, 7(1): 290-301. https://doi.org/10.47709/cnahpc.v7i1.5345

[16] Gaurav, K. (2021). Evaluation of XChaCha20-Poly1305 for improved file system level encryption in the cloud. https://norma.ncirl.ie/5937/1/kaholgauravbhuvanagiriudayakumar.pdf.

[17] Muhammed, R.K., Aziz, R.R., Hassan, A.A., Aladdin, A.M., Saydah, S.J., Rashid, T.A., Hassan, B.A. (2024). Comparative analysis of AES, Blowfish, Twofish, Salsa20, and ChaCha20 for image encryption. Kurdistan Journal of Applied Research, 9(1): 52-65. https://doi.org/10.24017/science.2024.1.5

[18] Pathak, A. (2023). Secured shared preferences in Android. Medium.

https://medium.com/@myofficework000/secured-shared-preferences-in-android-b4c2d8944c37.

[19] Encrypted-datastore. Github. from https://github.com/osipxd/encrypted-datastore.

[20] Muchamad, R.M., Asriyanik, A., Pambudi, A. (2023). Implementasi Algoritma advanced encryption standard (Aes) Untuk mengenkripsi datastore pada Aplikasi berbasis Android. Jurnal Mnemonic, 6(1): 55-64. https://doi.org/10.36040/mnemonic.v6i1.5889

[21] Klymenko, A., Karpenko, N., Gerasimov, V. (2025). Encryption and decryption of data in datastore for secure local storage. Системні Технології, 2(157): 187-196. https://doi.org/10.34185/1562-9945-2-157-2025-19

[22] Snaggy. (2024). Interacting with Android root privileges (rooted AVD) — Android CheatEngine like APP part 1. https://medium.com/@ahmadshamius2/interacting-with-android-root-privileges-rooted-avd-662f0ef4907a.

[23] Vohra, I. (2024). Preferences datastore is slow, but you should still choose it over Shared Preferences. Here's why…. https://proandroiddev.com/preferences-datastore-is-slow-but-you-should-still-choose-it-over-shared-preferences-heres-why-a5dc51c4a29e.