

SECURE CONFIDENTIAL FILE STORAGE USING ENCRYPTED SLICES ON MULTI-CLOUD

XU QIAN, Elankovan A. Sundararajan

Faculty of Information Science & Technology

Universiti Kebangsaan Malaysia

43600 Bangi, Selangor

Abstract

This research introduces SecureSlice, a new-generation solution addressing the paramount issue of secure cloud file storage. The OneSpace portion SecureSlice, is a more complete piece which stand out from other storage apps in that it utilizes strong encryption and advanced file slicing to store data spread across three primary consumer cloud platforms — Google Drive (GDrive), Dropbox, Box.

At the heart of SecureSlice is this concept — we protect your data in multiple layers. The content is turned into illegible files through heavy encryption before uploaded on any file sharing services. Then the encrypted file is split into several independent fragments. Without the decoding key, all three portions independently are useless, which also further mitigates against any threat of data leak. That way no one cloud service has all the pieces to rebuild the original file, and each of those sections are directly uploaded into user accounts on Google Drive, Dropbox and Box respectively.

Even if one of the cloud platforms is compromised, this only exposes a tiny encrypted block to an attacker that would be hopelessly impossible for them to read or perform any actions on. The file can only be recovered, reconstituted and decrypted by the original owner – on one side of this Selective Disclosure between CipherText Cloud/File Splitting system who has a decryption key that unlocks all three different clouds containing fragments. This storage system on a global scale is almost free of single point failures, and has improved the security

requirements at all levels far beyond simple data protection. Entity users which may be achieved.

However, SecureSlice is also user-centric by design. Files are owned and controlled entirely by users — you can seamlessly upload, manage or fetch sensitive documents without having to depend on any central service. The straightforward interface of the app makes authorization and file management easy to do, so that users no longer need knowledge about enterprise security.

SecureSlice combines encryption, segmentation and multi-cloud distribution to not only protect sensitive information from unauthorized access but ensure users can easily and securely store, share or publish their files anytime anywhere. This infrastructure defines a new standard for secure cloud storage as citizens of the digital world can experience security without sacrificing their rights to privacy and personal control.

1.0 INTRODUCTION

Cloud storage solutions have gained so much popularity as cloud computing technologies are being adopted rapidly by both individuals and organizations who find it very easy to store their data and access them in a quick span of time. Cloud platforms providing a flexible, scalable and easily accessible way to store data have transformed the landscape of storing & sharing data. That said, as this dependence on external cloud solutions has increased dramatically over the past decade it has equally introduced a slew of novel security threats chief among which are fears around unauthorized access to people's data and likely privacy violations more generally through such services (Subashini & Kavitha, 2011).

The project was created in response to the pressing questions outlined above, for a new and innovative application whose purpose is at its core reimagining how files are stored into/deleted/loaded- out of cloud based services sorting activities. Their solution makes sure that the user files are protected by strong encryption, — making it impossible to understand them if you do not know how to decrypt their content (Ali, Khan & Vasilakos, 2015). Rather

than uploading these single encrypted files, the system divides every file into numerous isolated fragments. After that, which means each encrypted fragment will mean nothing in and of itself; these are replicated among various independent cloud services a few big-name examples include Google Drive, Dropbox & Box(Kamara & Lauter, 2010).

This way, your data does not leak or someone access it and you are cool with in some risk that is reduced significantly. Even if one cloud provider is breached, a hacker could only access an encrypted fragment of the files and not reconstruct them into anything meaningful. They can only successfully pull back, bring together and unscramble the entire document with the proper client who has all cloud storage locations including all decryption keys. This kind of distribution in encrypted fragments makes the system much more secure and robust since it combines data confidentiality, integrity as well as user control to achieve a new level of cloud storage security(Kshetri, 2013).

2.0 LITERATURE REVIEW

Cloud Storage Security: Cloud storage services provide almost infinitely scalable and on-demand access to files but are also quite convenient for people—users can easily store data, getting vast amount of disk space and forgetting about this worry. But all these advantages, the price of which is attached via security in terms that include data leakage, loss and unauthorized access. Numerous cloud data breaches have brought to light the inherent risks of sensitive information being held on shared third-party platforms, particularly where control over such data is given up by users into providing hands.

Encryption Methods: End-to-end encryption is one of the most effective strategies to secure data in the cloud, ensuring that only authorized parties can access the contents of files. Despite its strengths, implementing robust encryption poses challenges in key management—users must securely store, share, and retrieve encryption keys, which is often cited as a major barrier to adoption in both personal and enterprise environments. Poor key management can result in permanent data loss or unauthorized disclosure.

File Fragmentation: Multiple studies have revealed data security can be further increased by splitting files into numerous pieces and storing different parts separately. As long as files are split over thirty storage locations the contents of a file will still be safe even if one location is compromised. Yet, file fragmentation creates challenges at the technical level — issues like proper restoring of fragments and synchronization between them efficiently across distributed cloud environments. It is still a challenge in research to securely and reliably treat with such fragmented data.

Multi-Cloud Storage: For instance, utilizing a multi-cloud storage strategy spreads files or file fragments across multiple independent providers (ex: Google Drive ,Dropbox and Box) to reduce the risk in dependence on any specific vendor. Should one provider be breached, only some of the data is at risk as the structure ensures that your other platforms are still secure. On the other hand, this approach adds complexity from a technical standpoint — with cross-cloud integration unified authorization and coordinated data management. Multi-cloud done right needs to solve for auth and sync at scale, mapped across multiple clouds user experience.

3.0 METHODOLOGY

3.1 User Needs

Amid exponential rise in sophisticated traps for cybersecurity and broader acceptance of the cloud-based services, a demand at all levels ever had arisen to weekly established but convenient approachings dedicated file storage. Nowadays users not only want their data to stay confidential but also wish to access the files across multiple devices and places in a seamless way, at any given hour.

Security Requirements:

The system must, above all, be secure and transparent from the end-user perspective – including ensuring that user files are always encrypted in transit as well. Before data is sent to an external platform it must be encrypted on the user’ s device so nobody (not even cloud

service providers) knows file content. Implementing strong encryption protocols to secure against unauthorized access and data breaches requires competent key generation mechanisms.

Fragmentation and Restoration:

For that, it is necessary to secure data fragmentation – dividing files into several parts before storing them on the system - and then from this, ensure a good response when users want their data retrieval. No one fragment ought to carry any context on its own, and only true assembly/de-cryption will yield a file. This mitigates the impact if a single storage provider suffers from a security breach.

Integration with Popular Cloud Platforms:

Given the dominance of major cloud storage providers such as Google Drive, Dropbox, and Box, the system needs to provide straightforward integration with these platforms. This includes support for industry-standard authentication (e.g., OAuth 2.0), efficient file transfer, and reliable cross-platform synchronization. Users should be able to link their accounts securely and manage their storage locations without technical barriers.

In conclusion, the main user requirements fulfillable by the future system are top-notch security with end-to-end encryption and slicing feature, excellent syncing capabilities of already widely-used cloud storage services (Amazon S3 in particular), as well good design which will facilitate perception a lot.

3.2 Conceptual Model Design

Architecture

The proposed architecture is a design concept, which aims to provide the maximum security of data together with flexibility in operation. The core workflow starts with

client-side file encryption; splits are then performed in two steps: finally, encrypted fragments are uploaded to a collection of completely separate cloud (bucket) nodes. This design of the workflow, in turn eliminates potential data exposure due to single-cloud dependency, creating a strong defense-in-depth model for cloud storage security.

Encryption

All user files are locally encrypted with the Advanced Encryption Standard (AES), specifically in Electronic Codebook (ECB) mode with PKCS5Padding, before any cloud interaction. This cryptographic mechanism guarantees that the content of file becomes totally gibberish for anybody who does not have appropriate decryption key. This is entirely on the user's device and ensures that not even cloud service providers can decrypt your stored data. This end-to-end encryption model improves the privacy and confidentiality of data in motion, mirroring secure cloud application industry best

practices.

```

package com.example.SecureSlice

import javax.crypto.Cipher
import javax.crypto.spec.SecretKeySpec
import java.security.SecureRandom
import android.util.Base64

object AesUtil {
    private const val AES_MODE = "AES/ECB/PKCS5Padding"
    private const val KEY_SIZE = 16

    // 生成随机密钥 (上传用)
    fun generateRandomKey(): ByteArray {
        return ByteArray(KEY_SIZE).also { SecureRandom().nextBytes(it) }
    }

    // 加密
    fun encrypt(input: ByteArray, key: ByteArray): ByteArray {
        val cipher = Cipher.getInstance(AES_MODE)
        val keySpec = SecretKeySpec(key, algorithm: "AES")
        cipher.init(Cipher.ENCRYPT_MODE, keySpec)
        return cipher.doFinal(input)
    }

    // 解密
    fun decrypt(encrypted: ByteArray, key: ByteArray): ByteArray {
        val cipher = Cipher.getInstance(AES_MODE)
        val keySpec = SecretKeySpec(key, algorithm: "AES")
        cipher.init(Cipher.DECRYPT_MODE, keySpec)
        return cipher.doFinal(encrypted)
    }
}

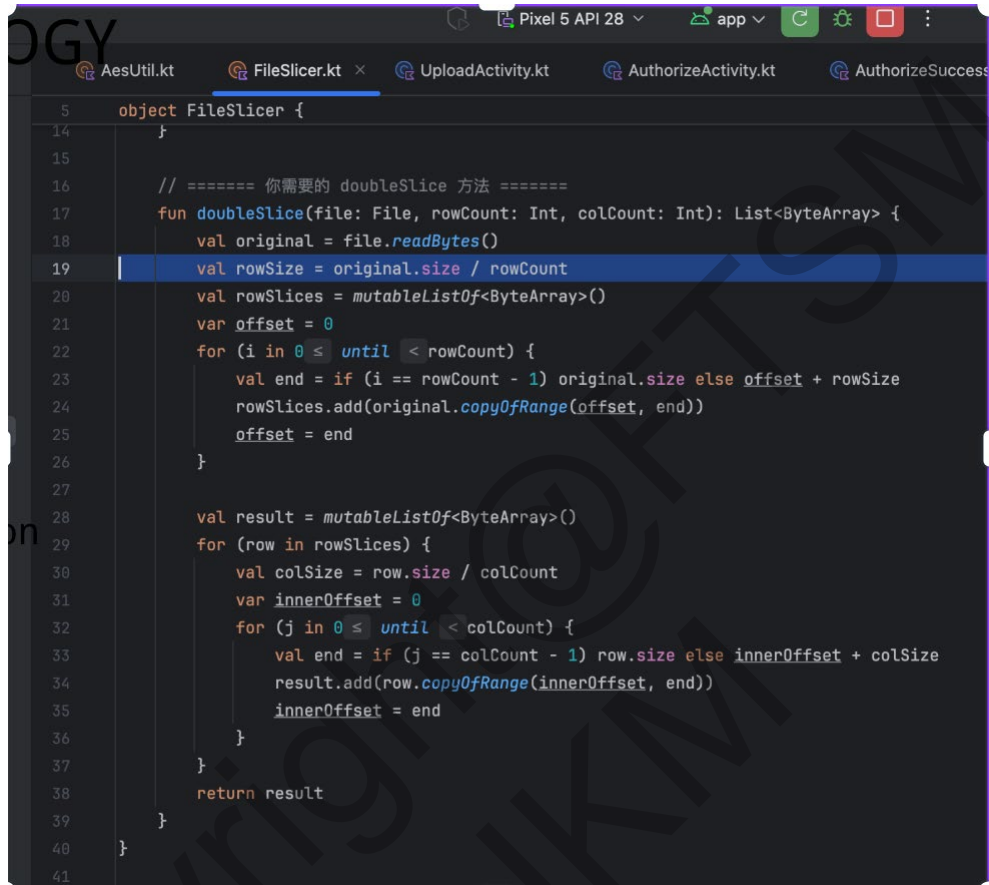
```

The core feature of the application is that it will first encrypt the user's file using a strong encryption algorithm, ensuring that the data is unreadable to anyone without the decryption key.

File Slicing

After encryption, each file undergoes a duplicitous slicing factor. In stage one, the file is horizontally partitioned— split by rows —and we get a set of tentative fragments. The extra subdivisions are separated by columns from further subdividing each of these segments so that we now have a grid made up of smaller 'chunks'. The ability to slice twice in this way changes the game as each resultant fragment can be both a fraction and an encrypted piece of data in the original file. No original files can be recovered unless the right key is used to decrypt them, and only by exactly recombining all fragments! While giving added support

towards data security this method can also bring redundancy as well as storage management flexibility.

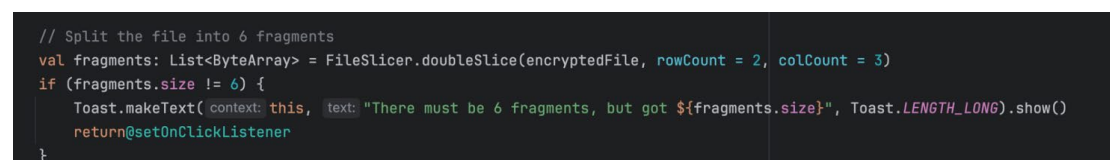


```

5  object FileSlicer {
14  }
15
16  // ===== 你需要的 doubleSlice 方法 =====
17  fun doubleSlice(file: File, rowCount: Int, colCount: Int): List<ByteArray> {
18      val original = file.readBytes()
19      val rowSize = original.size / rowCount
20      val rowSlices = mutableListOf<ByteArray>()
21      var offset = 0
22      for (i in 0 until rowCount) {
23          val end = if (i == rowCount - 1) original.size else offset + rowSize
24          rowSlices.add(original.copyOfRange(offset, end))
25          offset = end
26      }
27
28      val result = mutableListOf<ByteArray>()
29      for (row in rowSlices) {
30          val colSize = row.size / colCount
31          var innerOffset = 0
32          for (j in 0 until colCount) {
33              val end = if (j == colCount - 1) row.size else innerOffset + colSize
34              result.add(row.copyOfRange(innerOffset, end))
35              innerOffset = end
36          }
37      }
38      return result
39  }
40  }
41
  
```

We can first horizontally split the file which means each row will be considered as one of the fragments and then we vertically divide it by considering that number of frags per column so eventually, you are using 1..doubleSlice only calculate a total no. of fragments dynamically for given rowCount and colCount

Then defines how the file is splitted and calculates the total number of fragments as per given rowCount, colCount.



```

// Split the file into 6 fragments
val fragments: List<ByteArray> = FileSlicer.doubleSlice(encryptedFile, rowCount = 2, colCount = 3)
if (fragments.size != 6) {
    Toast.makeText(context, this, text: "There must be 6 fragments, but got ${fragments.size}", Toast.LENGTH_LONG).show()
    return@setOnClickListener
}
  
```


The doubleSlice method defines the way the file is split and dynamically determines the total number of fragments based on the given rowCount and colCount. However, the specific number of fragments is determined at the place where doubleSlice is called (such as in UploadActivity), according to the parameters passed in.

Cloud API Integration

The system is designed to easily integrate with the top cloud providers on the market today. Official Software Development Kits (SDKs) are used for Google Drive and Dropbox guaranteeing the functionality with their systems is stable, performant. They enable more secure upload/download of files, as well account management, leveraging OAuth2-based authentication flows for these features. If official SDK support for Box is limited or not ideal, we can integrate by interacting directly with the Box REST API and making HTTP requests (GET , POST etc.) using OkHttp library to send/receive data The modular nature of cloud connectivity and its components mean the system can easily move forward to support next-generation interfaces, as well as common regulatory considerations.

google	Official Google Drive SDK & OAuth2	<pre>implementation(libs.google.auth) implementation(libs.google.api.client) implementation(libs.google.api.services.drive) implementation(libs.google.http.client.android) implementation(libs.google.http.client.gson)</pre>
dropbox	Official Google Drive SDK & OAuth2	<pre>implementation("com.dropbox.core:dropbox-core-sdk:5.4")</pre>
box	Box is accessed directly via REST API calls using OkHttp.	<pre>implementation("com.squareup.okhttp3:okhttp:4.12.0")</pre>

4.0 RESULTS

Development Tools:

The application is developed using Android Studio as the primary IDE, with Kotlin as the programming language. Integration with cloud services is achieved using the official Google Drive SDK, Dropbox SDK, and direct interaction with the Box REST API.

Core Modules:

The system architecture is composed of several key modules:

Encryption/Decryption Module: Handles all AES encryption and decryption operations for file security.

Double Slicing Module: Implements the two-step slicing algorithm, splitting encrypted files horizontally and then vertically.

Cloud Upload/Download Module: Manages the transfer of file fragments between the app and multiple cloud storage providers.

Fragment Mapping Module: Keeps track of which fragments are stored on which clouds.

Key Management Module: Manages the generation, encoding, and secure handling of AES encryption keys.

Fragment Distribution:

Each six fragments per file Fragments 0 and 1 are in Google Drive, fragments 2 and 3 sit on Dropbox while Fragments V box. The logic behind this distribution is provided by the claim that no one cloud provider has enough data to rebuild the file.

```

package com.example.SecureSlice

class Assignment(
    val googleFragments: List<ByteArray>,
    val dropboxFragments: List<ByteArray>,
    val boxFragments: List<ByteArray>
)

object FragmentDistributor {
    fun distributeFragments(fragments: List<ByteArray>): Assignment {
        require( value: fragments.size == 6) { "There must be 6 fragments" }
        // 0,1 给 Google, 2,3 给 Dropbox, 4,5 给 Box
        val g = listOf(fragments[0], fragments[1])
        val d = listOf(fragments[2], fragments[3])
        val b = listOf(fragments[4], fragments[5])
        return Assignment(g, d, b)
    }
}

```

1) Distribute the 6 fragments evenly to 3 cloud drives in order, with each cloud drive getting 2 fragments.

2) Google gets fragments 0 and 1, Dropbox gets fragments 2 and 3, and Box gets fragments 4 and 5.

Fragment Naming:

Fragments are systematically named based on their destination cloud and their index position (e.g., gfrag_0.dat for Google Drive fragment 0), allowing for efficient retrieval and management.

```
// Dropbox (2,3)
assignment.dropboxFragments.forEachIndexed { i, fragment ->
    val name = "dfrag_${i}.dat"
    val indexInOriginal = indexMap[i + 2]
```

```
// Box (4,5), concurrent threads
val boxUploadThreads = mutableList0f<Thread>()
assignment.boxFragments.forEachIndexed { i, fragment ->
    val name = "boxfrag_${i}.dat"
```

```
// Google Drive (0,1)
assignment.googleFragments.forEachIndexed { i, fragment ->
    val name = "gfrag_${i}.dat"
    val indexInOriginal = indexMap[i]
```

Name ↑		Who can access	Modified
 dfrag_0.dat		☆ Only you	15/7/2025 23:49
 dfrag_1.dat		☆ Only you	15/7/2025 23:49
 gfrag_0.dat	Jul 15	 me	29 KB  My Drive
 gfrag_1.dat	Jul 15	 me	29 KB  My Drive
 boxfrag_0.dat		昨天由 XUQIAN	29.4 KB
 boxfrag_1.dat		昨天由 XUQIAN	29.4 KB

Key Management:

The AES encryption key is encoded in Base64 format to facilitate secure storage, user copying, and easy input during the file restoration process.

```

fun encrypt(input: ByteArray, key: ByteArray): ByteArray {
    val cipher = Cipher.getInstance(AES_MODE)
    val keySpec = SecretKeySpec(key, algorithm: "AES")
    cipher.init(Cipher.ENCRYPT_MODE, keySpec)
    return cipher.doFinal(input)
}

// 解密
fun decrypt(encrypted: ByteArray, key: ByteArray): ByteArray {
    val cipher = Cipher.getInstance(AES_MODE)
    val keySpec = SecretKeySpec(key, algorithm: "AES")
    cipher.init(Cipher.DECRYPT_MODE, keySpec)
    return cipher.doFinal(encrypted)
}

// Base64 编码密钥 (显示密钥给用户)
fun encodeKeyToBase64(key: ByteArray): String {
    // 移除任何可能多余的回车换行
    return Base64.encodeToString(key, Base64.NO_WRAP).replace(oldValue: "\n", newValue: "").replace(oldValue: "\r", newValue: "").trim()
}

// Base64 解码密钥 (从用户输入)
fun decodeKeyFromBase64(keyStr: String): ByteArray {
    // 去掉所有不可见字符
    return Base64.decode(keyStr.replace("\\s".toRegex(), replacement: ""), Base64.NO_WRAP)
}

```

1) Encryption and Decryption Methods

- `encrypt(input, key)`: Encrypts the input data using the provided key with AES, and outputs the ciphertext.
- `decrypt(encrypted, key)`: Uses the same key to decrypt the ciphertext and restores the original file content.

2) Key Base64 Encoding/Decoding

- `encodeKeyToBase64(key)`: Converts the binary key into a Base64 string so that users can save it.

`decodeKeyFromBase64(keyStr)`: Converts the user-input Base64 string key back into a binary key for decryption.

4.1 Application Evaluation

i. Functional Testing

Using exhaustive functional tests, we show that the application consistently performs well for all core functions within stestsuite mould: file encryption , double slicing, moving slice to cloud , getting slices from cloud 'n', combining them and decrypting back final file. All stages of the workflow were extensively prototyped with various types and sizes of input files to test for robustness. Without the correct AES encryption key, no restore is possible and requires unlocking before any data can be read. If a wrong key is passed, the application locks down access inhibiting unauthorized viewing of files essentially protect user information!

ii. Usability Testing

Startup and Welcome Screen

When the user opens the SecureSlice app, they are greeted with a clean welcome page. The main section displays the app icon and name, with a prominent "Get Started" button below.

The user taps "Get Started" to proceed to the next step.



User Login/Registration

If the user already has an account, they can enter their email and password and tap "Login" to sign in.

If the user is new, they can tap "Don't have an account? Register" at the bottom to switch to the registration page. The registration page asks the user to fill in their username, email, and password. After completing the form and tapping "Register," registration is completed.

After successful registration, the system automatically returns to the login page and displays a "Registered successfully" message. The user can now log in with their newly created account

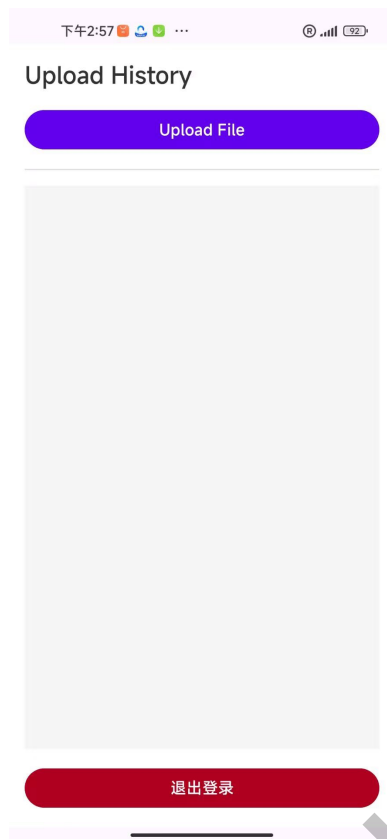


Main Interface – Upload History

After logging in, the user enters the main "Upload History" page, where they can view the record of files uploaded in the past (which is empty upon first use).

At the top of the page, there is an "Upload File" button that the user can tap to start uploading a new file.

At the bottom of the page, there is a "Logout" button, allowing the user to log out at any time



Cloud Authorization Process

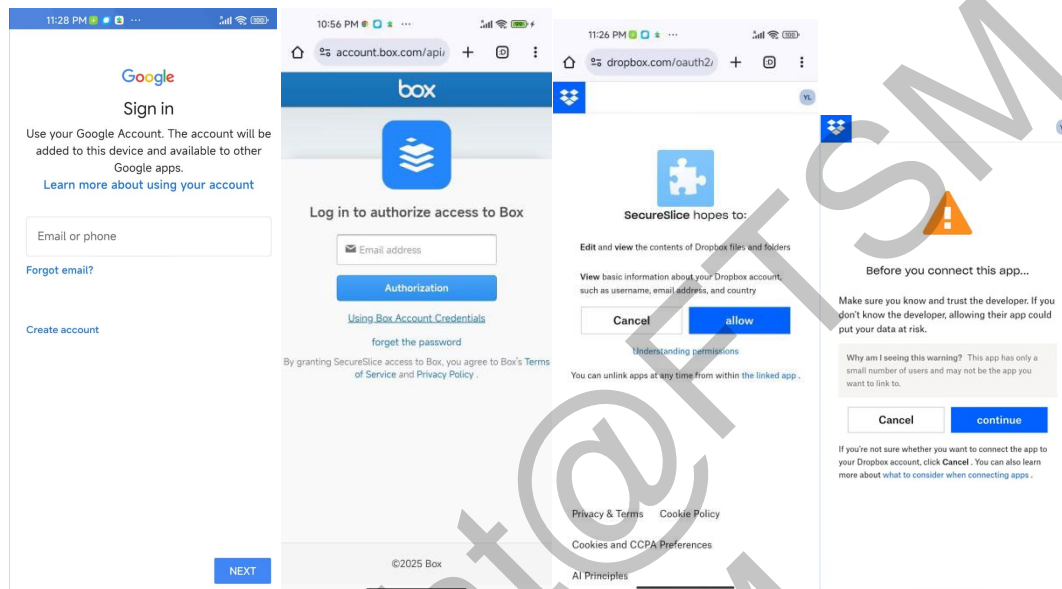
Before uploading files, the system will prompt the user to authorize Google Drive, Dropbox, and Box cloud accounts individually.

When authorizing Google Drive, the system brings up the Google account selection screen, where the user selects their Google account and grants permission.

When authorizing Dropbox, the app jumps to the Dropbox OAuth2 authorization page. The user taps "allow" to grant SecureSlice access to their Dropbox files.

When authorizing Box, the user needs to enter their Box account and password, and then tap "Authorization" to grant access.

During the authorization for all platforms, users do not need to provide their passwords to SecureSlice; all sensitive information is entered only on the official authorization screens, ensuring security.



File Upload

After authorization is complete, the user can tap "Upload File" to select a local file to encrypt and upload.

The app will automatically perform file encryption and double slicing, then upload the six resulting fragments to Google Drive, Dropbox, and Box, respectively.

After the upload is complete, the system will display an upload success message and generate an AES key (Base64-encoded) for subsequent file restoration.



Upload Successful!

nOiDkSEGLmyWZX2mRQR/yg==

Copy Key

Back

Continue Upload

✓ Google上传成功: gfrag_1.dat

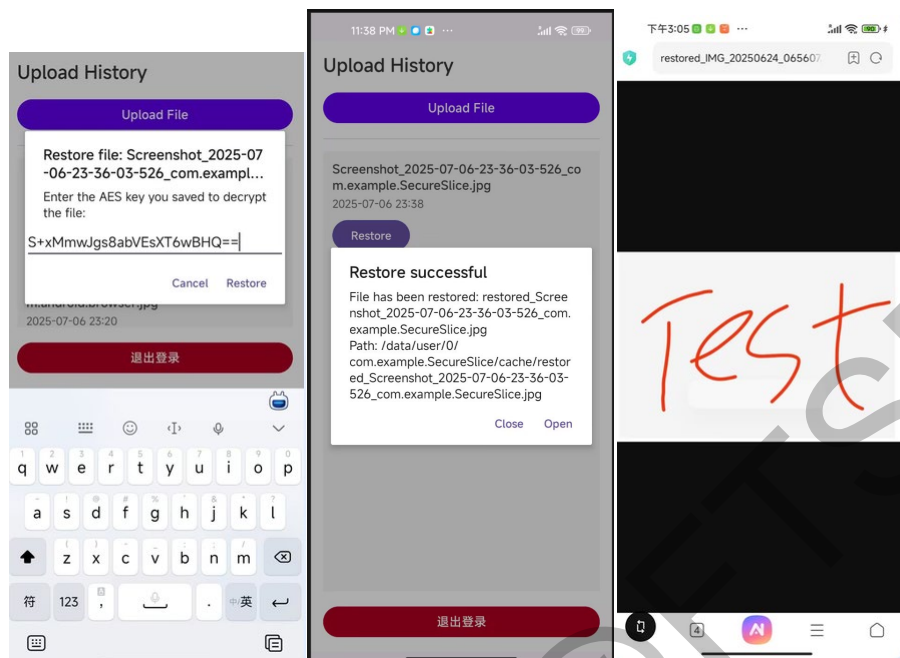
File Download and Restoration

Users can view all upload records on the main "Upload History" page.

To restore or download a file, the user taps the relevant record, and the system will prompt for the AES key.

Only when the correct key is entered can the app automatically download the file fragments from the respective clouds, merge and decrypt them, and restore the original file.

If the key is entered incorrectly, the file cannot be restored, ensuring data security.



5.0 CONCLUSION

This project provides secure and confidential file storage in multiple clouds based on encryption, fragmentation along with distributed storage. SecureSlice also offers users access to total control over their data, thereby drastically reducing possibilities for unauthorized use and security breaches. The app is essentially a new solution not available in the market..

6.0 APPRECIATION

I would like to express my gratitude to my supervisor, Elankovan A. Sundararajan, for his guidance and support throughout this project. I also thank my peers and family for their encouragement.

7.0 REFERENCES

Subashini, S., & Kavitha, V. (2011). A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1), 1-11.

Ali, M., Khan, S. U., & Vasilakos, A. V. (2015). Security in cloud computing: Opportunities and challenges. *Information Sciences*, 305, 357-383.

Kamara, S., & Lauter, K. (2010). Cryptographic cloud storage. In *Financial Cryptography and Data Security*, 136-149.

Kshetri, N. (2013). Privacy and security issues in cloud computing: The role of institutions and institutional evolution. *Telecommunications Policy*, 37(4-5), 372-386.

XU QIAN (A191511)

Dr. Elankovan A. Sundararajan

Faculty of Information Technology & Science

National University of Malaysia