The Cloud:
Is it Safe Enough to Store Files?
Cloud Storage is amazing… and risky

The 2000s may be over, but cloud storage is still amazing. Services such as Dropbox, Apple's iCloud, Google's Drive, and Microsoft's OneDrive all help users share files with friends, recover when a hard drive crashes, and move files between their devices. Still, hearing about data breaches[1, 2, 3] leaves people wondering whether their data is safe in the cloud.

Security on today's cloud lacks end-to-end-encryption

Today's cloud storage services use fairly similar client–server architectures that start with a locally installed application that monitors a specific file folder on a user's computer. When the app detects changes in the folder, it relays them to the user's account on the cloud. The cloud service handles copying them to a user's other devices.

One of the problems with cloud storage is that files are not end–to–end encrypted (E2EE) – meaning that files encrypted before leaving a user's device do not remain encrypted until they return. Rather, most providers use the transport encryption + encryption at rest paradigm. In this model, transport encryption encrypts files sent to the server (e.g., HTTPS), but decrypts them upon arrival. Next, the server applies encryption at rest so that only encrypted files are stored. While providers tout the strength of their encryption algorithms (e.g., AES 256), what they don't highlight is that the server decrypts user files before re–encrypting them and that they hold the decryption keys!

Figure 1 shows how this process works:

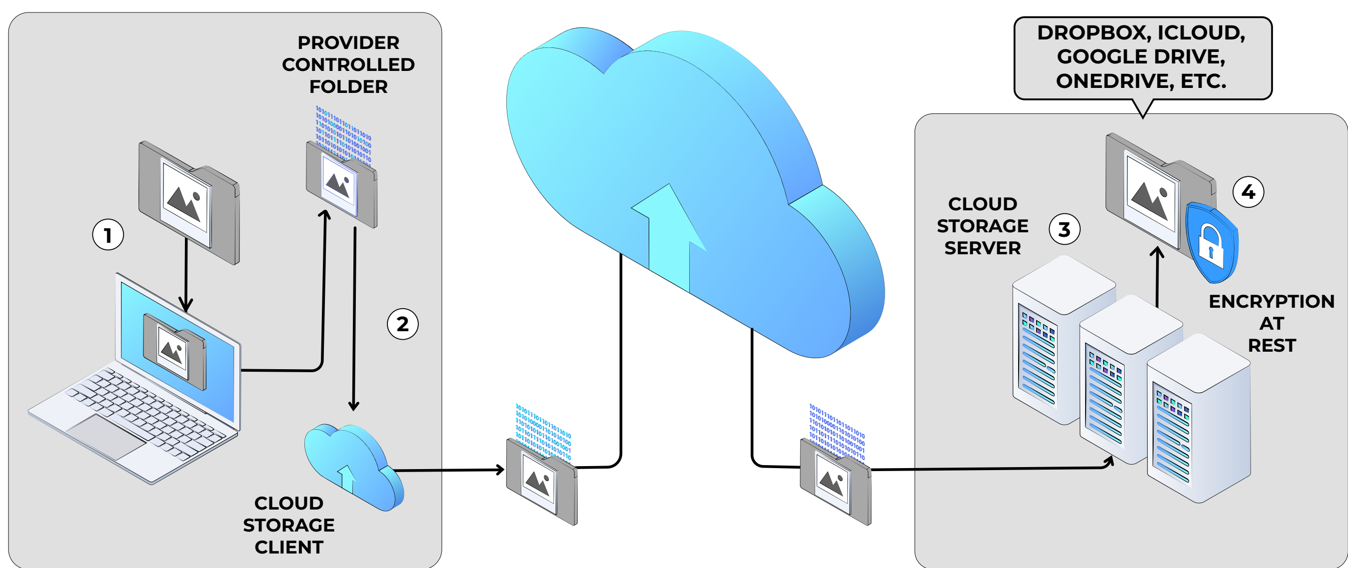


Figure 1 - Cloud storage overview

While this model is efficient, it is vulnerable to attack and unfortunately is the mainstay of cloud storage. This leaves users wondering: Are cloud providers accessing my data? and Can hackers steal my digital files?

We can make the cloud safer

Answering those questions is difficult and varies over time. So, how do individual users protect their files in the cloud? One idea is to help users layer end–to–end encryption (E2EE) on top of any features the cloud storage services provide. This is fairly easy using the cryptographic features of decentralized identity (DI).

Store (for a long time) **and forward** (later)

Internet communications are based on the store and forward paradigm. In this model, when a user sends a message, it is divided into packets that are relayed between message servers (sometimes waiting at each relay) until they reach their destination. This is a good model and can be applied to file storage.

One of the advancements introduced by DI is a specification known as DIDComm. Much like the internet's store and forward paradigm, DIDComm defines a method of applying E2EE to encrypt messages between a sender and a receiver. E2EE ensures that DIDComm messages are never decrypted to plaintext except by the designated recipient. DIDComm messages are formatted using JavaScript Object Notation (JSON) with header information that denotes sender and receiver along with an encrypted message body. Figure 2 shows a simple DIDComm message:

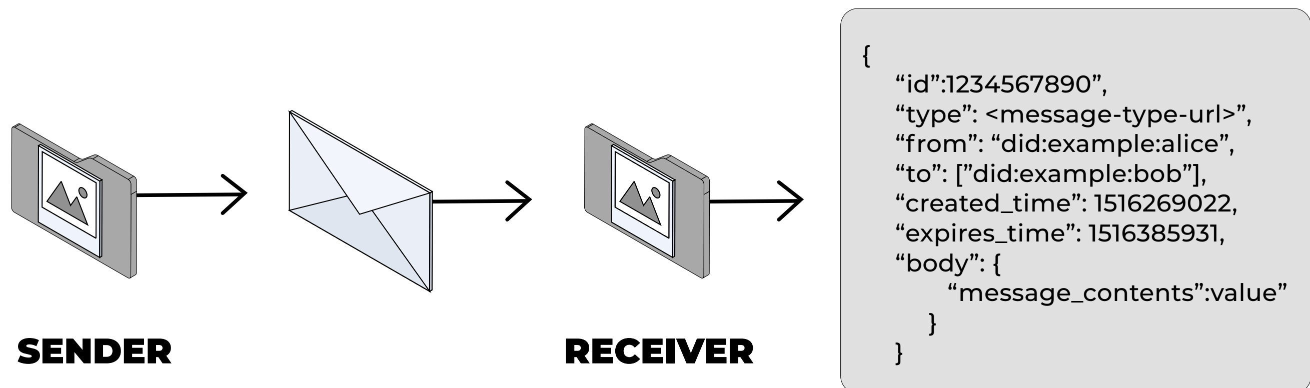


Figure 2 - Simple DIDComm message

In the DIDComm model, if a message is sent by User A and is only addressed to User A, then it can only be decrypted by User A—no matter where the message travels. This capability provides a good basis for securing provider–independent cloud storage.

Here's a high-level overview of the encrypt – store – decrypt process:

1. User A: encrypts a file (addressed to User A) using DIDComm
2. Cloud Storage: receives and stores the DIDComm message (i.e., E2EE)
3. User A: sometime later retrieves the file and decrypts it.

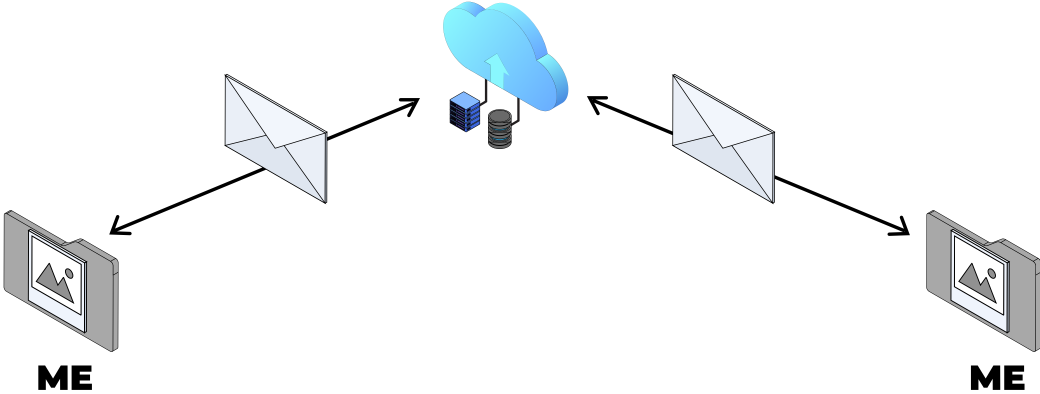


Figure 3 - Encrypt - store - decrypt process

Given the E2EE benefits of DIDComm messaging, the fully encrypted message files may be safely stored on the provider's cloud storage. This process resembles an enhanced version of standard internet communications: store (for a long time) and forward (later).

What is a DID?

In Figure 3, the encrypt–store–encrypt model is illustrated by a user sending a delayed delivery message to themself. We might playfully call this me–to–me communication in order to highlight that being both the sender and receiver of messages lets us use modern technologies in tangential ways that have significant benefits.

Even though the Me shown in Figure 3 depicts a user sending a message to themself, the DI protocols require a user to identify themself using a decentralized identifier (DID). There is an entire specification describing DIDs, but generally they resemble web URLs:

Figure 4 - Comparing DIDs to URLs

While DIDs can come from many different providers, the simplest form of a DID is one that users (or their apps) create for themselves. This type of DID is called a "DID key" and denoted by the prefix did:key. Using the example from Figure 4, a DID key takes the form of did:key:123456789abcdefghijk where the latter element is simply an encoded version of the public key that the user has created.

Usability comes from extending the cloud storage synchronization method

While the encrypt – store – decrypt process presented in Figure 3 is straightforward, automating it will make it very simple for users. Automating this process will leverage the same synchronization model that today's cloud storage utilizes.

In contemporary cloud storage synchronization, the cloud storage service installs a cloud storage client application on user's device. This client application monitors a specified directory on a user's device and watches for changes to its file content. Those changes are synchronized with the cloud storage and replicated out to a user's other devices.

Instead of applying the encrypt – store – decrypt process to the cloud exactly as depicted in Figure 3, its concepts will be used to create an enhanced process that will synchronize (while encrypting/decrypting) files between a new user's plaintext folder to be created on the user's local system (e.g., /Users/username/PlaintextFiles/) and the folder monitored by the cloud storage provider (e.g., /Users/username/Dropbox/).

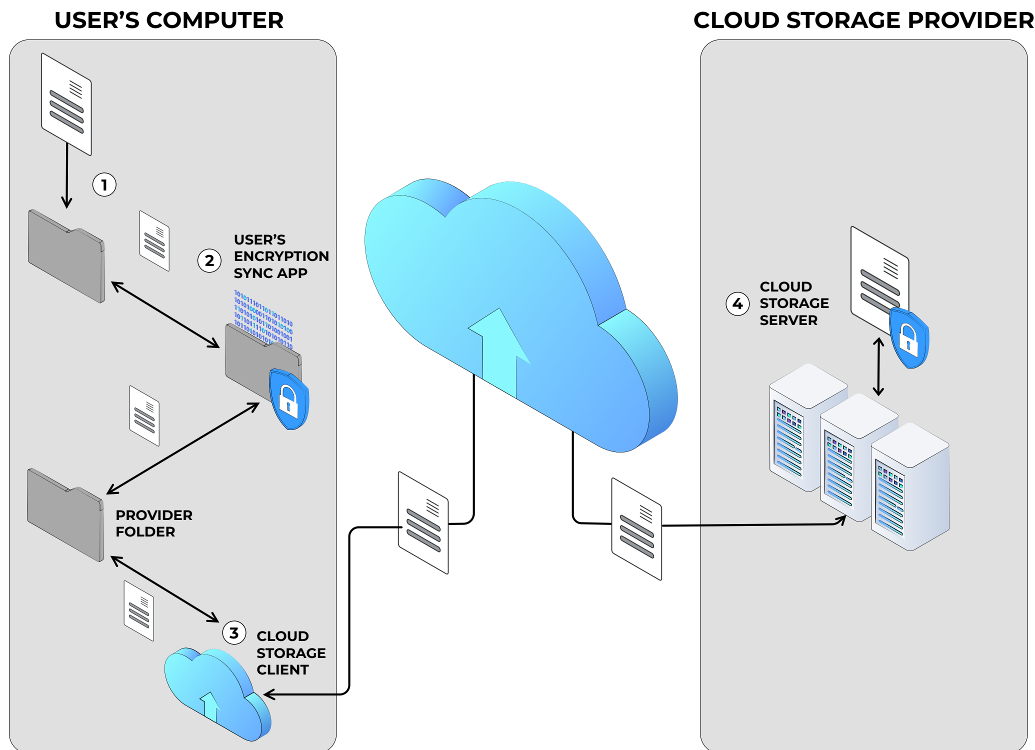


Figure 5 - Extended cloud storage synchronization method

The process described in Figure 5 (above) illustrates two synchronization processes. The first is performed by the cloud provider's Cloud Storage Client application, which remains unaltered and operates as intended – namely, to synchronize files between the Provider Folder and the cloud.

The second synchronization process depicted in Figure 4 (above) is the new User's Encryption Synchronization Application. This application monitors for changes (e.g., additions, deletions, modifications) in the User's Plaintext Folder and also changes in the Provider Folder. When additions or deletions are detected in the User's Plaintext Folder, those files are encrypted and copied to the Provider Folder. When a file is deleted from the User's Plaintext Folder, its encrypted version is deleted from the Provider Folder. Similarly, when an encrypted file is added to or modified in the Provider Folder, it is decrypted and copied to the User's Plaintext Folder. When encrypted files are deleted from the Provider Folder, their unencrypted variants are deleted from the User's Plaintext Folder.

This process has several benefits:

1. **Simple**

   Uses only local file synchronization; no network communication.

2. **API independent**

   Does not use any cloud provider APIs.

3. **Interoperable**

   Can synchronize files to any cloud provider's folder.

4. **Standardized**

   Using DIDComm leverages open-source standards for encrypted files.

5. **Algorithm choice**

   Users may change encryption algorithms as desired.

6. **Key management**

   Users maintain their own keys (in a DI wallet) and don't need to worry about how a cloud provider might protect them.

Figure 6 - Benefits of synchronizing the encrypt - store - decrypt process

Encrypted file format

DIDComm messages are formatted as plaintext JSON structures. For cryptographic security, the sensitive elements of a DIDComm message are encrypted using a selectable set of encryption methods, such as AES256-GCM, AES256-CBC with an HMAC-SHA512, etc. Currently, the available elliptic curves include X25519, P-384, P-256, and P-521. As quantum encryption methods are finalized, it is foreseen that those algorithms will be quickly adopted into a future iteration of the formal DIDComm Messaging Specification.

A detailed description of the DIDComm message format is left to the [DIDComm Messaging v2.0]() specification.

However, Figure 7 (below) has one particular field that is of particular note to this process. The recipient's name field contains a name field called header, which has a member kid. In DIDComm, kid contains the recipient's public key which was used to encrypt the ciphertext. While the DID specified in this field can use a wide range of DID methods, for the purposes of this use case, a simple did:key is used. Since the user is creating this DIDComm message to be sent (or stored) only to themself, using a did:key that represents a keypair already contained within the user's DI wallet is the simplest option.

For reference, Figure 7 depicts an encrypted DIDComm message:


```
{
        "protected":"eyJ0eXAiOiJhcHBsaWNhdGlvbi9kaWRjb21tLWVuY3J5cHRlZCtqc29uIiwiZW5jIjoiQ
        TI1NkdDTSIsImtpZCI6IldfVmNjN2d1dmlLLWdQTkRCbWV2VnctdUpWYW1RVjVyTU5RR1V3Q3FsSDAiLCJza2lkIj
        oiZGlkOmtleTp6Nk1raVRCejF5bXVlcEFRNEhFSFlTRjFIOHF1RzVHTFZWUVIzZGpkWDNtRG9vV3AiLCJhbGciOiJ
        FQ0RILTFQVStBMjU2S1cifQ",
        "recipients":[
                {
                        "header":{
                                "key_ops":[],
                                "alg":"ECDH-1PU+A256KW",
                                "kid":"did:key:z6MkiTBz1ymuepAQ4HEHYSF1H8quG5GLVVQR3djdX3mDooWp",
                                "epk":{
                                        "kty":"OKP",
                                        "crv":"X25519",
                                        "x":"sI4GXfe7pHrZ1I3jhB_UT34Iw5IMvgf8npd2UwLUe1I"
                                },
                                "tag":"MyYLSqwL-zV5p4KIG6hfaQ",
                                "iv":"z77anoWbR0v7Ihec"
                        },
                        "encrypted_key":"XkWFukZDBuz2Jx82w1uvpM6Pl6T9XJJQg3Bx9x4AV2w"
                }
        ],
        "ciphertext":"2Rhl0fvf1frZI_hUhY2-
        O7ld9sWphf5XCMOY1ebqSmZjff_FyC19FrKoxRAYQueyHn9jwU
        -u1qC0bTFb17YFu0ViFmx0SsIkL1ntyJlurpxoWs4eqkil2BHrriyW2bC8aUp5vrstenG8gckAA4jq57x_HoL2v8ry
        hLyvOUkrza2Pk0W0P8afIE99F74A0Feno47aganUD5Jg_OlxC07h4Zkqv-
        qltyoaf6ot0IadEc7e4AdQUdriEliacxZMMb87vzudFv4-5DrNCTfSdOqP-UqlzyJYjgoJkYD81cGjC-
        yvTPJVU5_msfOD4n6DXU0JKgPdU0R6KoyTBhiB8Arr_i5kgzaQVcq0roiHEp04gJrYeMH6jHsSPIh7CNCMz3U3KMJ
        _wFMjSDnKiDLFU3g2SFC2IpWuBrgCfII_KvftFgIMEtXvgd-iX1QT1B6TkGWyhgnZUsmt8FXDrBGuNgRbvPoOZk-
        ewol0eq5dRI5IoVI9GQU5oxIyImKTcoi00mPqtyWyXc_CfRrJXKRUANUtjefonKeQFtieqh4451e_Fgd10UGPY3yl
        L82krZtGh-QLqxqXqDMuzvw29hsYOdmYgr8m-
        dCWcaULW6lLN69QDVOnT4cin1n2Q0QGGnisS__VobkzYVBAHd_30NyvzgXjuCFXmHho9LM-
        CTjv92JmmZrZFWIFAEsqK5zB24bBfW7A1Ouf7FKo4TfXLr_z17amyDVpxL27U6b4yMbXuPemNUk7-
        z7km7NS8ffwZcKZko-
        R4NOlf56RZ1ZELTWhqgrfF_X7FzTpyJj7PzAI9zuNSzliIHRW7qOTdT_CcBFmrXiJGUNrLmI5C74PcctrdVLC93YR
        hfNVsGWDaKtQbnnXg3WQirVwDpELUludE-aXgeOHSjv7UhvnInEKglsNiCgSs40Kf06ToLDHpi1hwFBFoHwq-
        Ad5wrD_EOvMTvW8LmFSpru1",
        "iv":"2jBWPDXxzTcFWBVx_jGnTXf8p5ANi4rJ",
        "tag":"6wQn2QSRhnVq4qRUC-dP2Q"
}
```


Figure 7: Sample encrypted DIDComm message

Encrypted file naming

Creating a name for an encrypted file is not as simple as encrypting it. In part, this is due to some encryption algorithms having cyphertext output lengths that are different from their plaintext inputs. This could potentially result in encrypted filename lengths that are longer than the maximum that the host file system allows.

In order to avoid these types of situations, ciphertext file names are created as an HMAC256 encoding of the plaintext filename, which has been subsequently base58 encoded. HMAC256 provides sufficient cryptographic protection and ensures a limited text length, while the base58 encoding ensures that no incompatible characters are present in the resulting name string.

File management dilemma

Using the DIDComm message format greatly simplifies the implementation of extensible encrypted file storage. When the User's Encryption Sync App (described above in Figure 5), detects file changes in the User's Plaintext Folder, it will handle them as follows:

New file:

The detected file will be encrypted and copied into the Provider Folder. The name of the encrypted file in the Provider Folder will be an HMAC256 encoded version of the plaintext file name that has also been base58 encoded.

Modified file:

Modified files are handled essentially the same as new files with the addition that the existing encrypted file will be overwritten by the newly encrypted file.

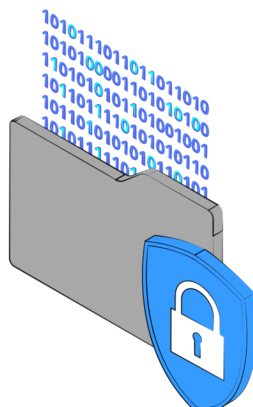
Deleted file:

When deleting a plaintext file, the corresponding encrypted file must be deleted from the Providers Folder. To do this, the plaintext filename is first encoded in the same way as for new files. The Provider Folder is searched for a file with the encoded name, which is deleted when found.

When the User's Encryption Sync App detects changes in the Provider Folder (i.e., containing the encrypted files), it will do the following:

New file:

Decrypt the encrypted file and create the corresponding plaintext file in the User's Plaintext Folder.



Modified file:

This operates essentially the same as with creating a new file. The encrypted file is decrypted, and the corresponding plaintext file is created in the User's Plaintext Folder. Any existing plaintext file is overwritten.

Deleted file:

When an encrypted file is deleted from the cloud storage provider's cloud, that file delete action is replicated on the user's device and results in the encrypted file being deleted from the Provider Folder. By the time that the User's Encryption Sync App detects that an encrypted file has been deleted, it has already been deleted. This means that the filename cannot be decrypted (using any information contained within the file) to know which corresponding plaintext file to delete. This situation requires special handling.

Adding a new file header

Handling the case where encrypted files are deleted before their plaintext counterparts (as described above) is best addressed by adding a new file header that will be prepended to the DIDComm message in the encrypted file. Figure 8 depicts the new file header:

```
{
        "encrypted_filename":"Y6MNigbxeQHUbJgjWQWJpv",
        "filename_hmac256":"8JAFm29SoTanBEkEdQgNnYMUn1xPuo6mLrLrFt7EFHo4",
        "key_id":"11111111111111111111111111111111"
}
```

Figure 8 - Encrypted File Header

The file header has three members:

1. **encrypted_filename:** an AES256 encryption of the plaintext filename with the encrypted output being base58 encoded
2. **filename_hmac256:** an HMAC256 computation of the plaintext filename with the output being base58 encoded
3. **key_id:** a textual identifier of the AES encryption key that is created and is contained within the user's DI wallet.

This header implements the functionality described above. It also allows the easy decryption and recognition of the target plaintext file.

Deleting the plaintext file when the encrypted file has already been deleted

When deleting a plaintext file associated with an encrypted file that has already been deleted, it is necessary to search for the plaintext file since it is not possible to directly identify it. The algorithm for locating the plaintext file is depicted in the following Rust language code:

```
fn plaintext_file_associated_with_encrypted_file(
        encrypted_filename: String,
        source_root: String,
        dest_root: String,
        key_id: [u8; AES256_KEY_LENGTH]) -> String {

        let mut file_to_delete: String = "".to_string();

        // Scan the plaintext directory. For now, this only scans the directory.
        // and not sub-directories. Eventually, this will scan the entire sub-directory
        // tree.
        let paths = fs::read_dir(dest_root.clone()).unwrap();
        for path in paths {
                if let Ok(p) = path {

                        // Get a String representation of the path.
                        let filepath = p.path().clone().into_os_string()
                        .into_string().unwrap();

                        // Remove the full path component to get the relative path portion.
                        let file_rel_path = filepath.replace(&dest_root, "");

                        // Create a hash of the relative file name and create mock-up of a
                        // would be encrypted file.
                        // This enables the calculated_encrypted_filename to be compared
                        // with the actual encrypted_filename that was deleted.
                        let hashed_filename = hash_filename(file_rel_path.clone(),
                        key_id.to_base58().clone());
                        let calculated_encrypted_filename = source_root.clone() +
                        &std::path::MAIN_SEPARATOR_STR.to_owned() +
                        &hashed_filename;
                        if encrypted_filename.to_string() ==
                                calculated_encrypted_filename.to_string() {
                                file_to_delete = filepath;
                                break;
                        }
                }
        }

        return file_to_delete;
}
```

Figure 9 - Calculating the plaintext filename associated with an encrypted filename

The method described above searches for the target plaintext filename by encrypting and base58 encoding each file in the directory and comparing that with the encrypted filename that was reported as deleted. While this method is not as fast as directly decrypting an encrypted filename, it handles the case of needing to match an existing plaintext file to an encrypted file that is no longer available.

With pretty printing applied, the full DIDComm encrypted message with the added file header (which is what is written to the encrypted files) will appear as follows:


```
{
        "encrypted_filename":"Y6MNigbxeQHUbJgjWQWJpv",
        "filename_hmac256":"8JAFm29SoTanBEkEdQgNnYMUn1xPuo6mLrLrFt7EFHo4",
        "key_id":"11111111111111111111111111111111"
}
{
        "protected":"eyJ0eXAiOiJhcHBsaWNhdGlvbi9kaWRjb21tLWVuY3J5cHRlZCtqc29uIiwiZW5jIjoi
        QTI1NkdDTSIsImtpZCI6IldfVmNjN2d1dmlLLWdQTkRCbWV2VnctdUpWYW1RVjVyTU5RR1V3Q3FsSDAiLCJza2lkIj
        oiZGlkOmtleTp6Nk1raVRCejF5bXVlcEFRNEhFSFlTRjFIOHF1RzVHTFZWUVIzZGpkWDNtRG9vV3AiLCJhbGciOiJ
        FQ0RILTFQVStBMjU2S1cifQ",
                "recipients":[
                {
                        "header":{
                                "key_ops":[],
                                "alg":"ECDH-1PU+A256KW",
                                "kid":"did:key:z6MkiTBz1ymuepAQ4HEHYSF1H8quG5GLVVQR3djdX3mDooWp",
                                "epk":{
                                        "kty":"OKP",
                                        "crv":"X25519",
                                        "x":"sI4GXfe7pHrZ1I3jhB_UT34Iw5IMvgf8npd2UwLUe1I"
                                },
                                "tag":"MyYLSqwL-zV5p4KIG6hfaQ",
                                "iv":"z77anoWbR0v7lhec"
                        },
                        "encrypted_key":"XkWFukZDBuz2Jx82w1uvpM6Pl6T9XJJQg3Bx9x4AV2w"
                }
        ],
        "ciphertext":"2Rhl0fvf1frZI_hUhY2-
        O7ld9sWphf5XCMOY1ebqSmZjff_FyC19FrKoxRAYQueyHn9jwU-
        u1qC0bTFb17YFu0ViFmx0SsIkL1ntyJlurpxoWs4eqkil2BHrriyW2bC8aUp5vrstenG8gckAA4jq57x_HoL2v8ry
        hLyvOUkrza2Pk0W0P8afIE99F74A0Feno47aganUD5Jg_OlxC07h4Zkqv-
        qltyoaf6ot0IadEc7e4AdQUdriEliacxZMMb87vzudFv4-5DrNCTfSdOqP-UqlzyJYjgoJkYD81cGjC-
        yvTPJVU5_msfOD4n6DXU0JKgPdU0R6KoyTBhiB8Arr_i5kgzaQVcq0roiHEp04gJrYeMH6jHsSPIh7CNCMz3U3KMJ
        _wFMjSDnKiDLFU3g2SFC2IpWuBrgCfII_KvftFgIMEtXvgd-iX1QT1B6TkGWyhgnZUsmt8FXDrBGuNgRbvPoOZk-
        ewol0eq5dRI5IoVI9GQU5oxIyImKTcoi00mPqtyWyXc_CfRrJXKRUANUtjefonKeQFtieqh4451e_Fgd10UGPY3yl
        L82krZtGh-QLqxqXqDMuzvw29hsYOdmYgr8m-
        dCWcaULW6lLN69QDVOnT4cin1n2Q0QGGnisS__VobkzYVBAHd_30NyvzgXjuCFXmHho9LM-
        CTjv92JmmZrZFWIFAEsqK5zB24bBfW7A1Ouf7FKo4TfXLr_z17amyDVpxL27U6b4yMbXuPemNUk7-
        z7km7NS8ffwZcKZko-
        R4NOlf56RZ1ZELTWhqgrfF_X7FzTpyJj7PzAI9zuNSzliIHRW7qOTdT_CcBFmrXiJGUNrLmI5C74PcctrdVLC93YR
        hfNVsGWDaKtQbnnXg3WQirVwDpELUludE-aXgeOHSjv7UhvnInEKgIsNiCgSs40Kf06ToLDHpi1hwFBFoHwq-
        Ad5wrD_EOvMTvW8LmFSpru1",
                "iv":"2jBWPDXxzTcFWBVx_jGnTXf8p5ANi4rJ",
                "tag":"6wQn2QSRhnVq4qRUC-dP2Q"
}
```


Figure 10: DIDComm encrypted file with file header

Additonal use cases are around *survivable encryption*

Selecting the DIDComm encrypted message format for protecting files stored on cloud storage platforms has resulted in several benefits as described. However, since DIDComm was designed as a messaging format, it may enable a wide range of additional use cases that could be built on top of the encrypted file storage architecture that has been presented.

One additional use case is survivable encryption. Many people like to keep daily journals, document their inventions, record financial account activity, keep cryptographic wallets, etc. and want to backup those data stores to the cloud in order to protect against a wide range of disasters. Even the most privacy-centric users may want to have the option of keeping their data protected while they are alive and then having it securely delivered to their heirs, some day.

Some data survivability services will hold user data and deliver it to designated heirs upon verification of a user's death. This is a valuable service, but it usually implies that the service will have control of the unencrypted versions of the data being preserved. Even if the survivability service has a privacy policy, it is of little comfort to the security conscious when such policies say (in effect), "You can trust us, because we have a privacy policy ..." or "Your data is safe, because we promise to ...".

For privacy-oriented users, the DIDComm specification enables them to take direct control of their data's security. Since DIDComm inherently facilitates multiple recipients, a user could setup their User's Encryption Synchronization App (as described above) to archive their files, not only addressed to themself, but also addressed to anyone they designate as an heir. Doing this leaves each file in an undelivered state where both the user and their heir(s) can, upon receipt, decrypt the files at some future date.

Instead of maintaining control of the plaintext data, the survivability service would only need to know when and upon what condition (e.g., death certificate) they are instructed to release the DIDComm encrypted files and then be able to transmit the encrypted data files as previously specified by the user (in the DIDComm message). This method enables users to select their own heirs while contracting with a survivability service that will perform the posthumous file delivery without the service being able to access any of the plaintext data content.

Final answer to our first question? E2EE messaging capability would make the cloud safer

Cloud storage is an amazing invention and delivers a wide range of measurable benefits, but in its current form, most cloud storage architectures leave the data entrusted to them somewhat vulnerable. The DIDComm messaging specification was created to provide a platform-independent yet interoperable encrypted messaging capability that enables users of a wide range of DI platforms to exchange end-to-end encrypted messages. This E2EE messaging capability can be used to secure files stored on virtually any cloud storage platform without divulging any plaintext file data content to the cloud service. Further, by storing secure files in an encrypted messaging format, those files can potentially be activated to later perform a myriad of secure file services for security and privacy conscious users.

Note

A cloud storage encryption service based on the concepts described in this whitepaper has been reduced to practice as an open-source tutorial. Both the detailed tutorial and the corresponding source code (Rust and Java) can be found at: https://github.com/sudoplatform-labs/protecting-cloud-storage