CertaCrypt File Manager – Capability Based P2P Filesystem App

The core component of the master thesis is a programming library ("CertaCrypt") that can be used in many ways. As a demonstration for its abilities, but also for defining requirements and to support internal design choices, an application will be built on the top of that library.

From a novice user's point of view the application is just another cloud file storage with the ability to share individual files or directories with other users and giving other users write permissions to files or directories. Similar alternatives would be Dropbox, Google Drive or Microsoft OneDrive.
Here, the main distinction is that a user does not (necessarily) have to sign up for the service and that the user does not need to trust a provider to keep the files confidential.

Users should be able to share read access by "inviting" individual users, as well as by sending an URL over other channels. This URL not only references to the shared data, but also already includes an encryption key for decrypting its contents. Files can also be set to "public", this way they are not encrypted and therefore readable for everyone that has knowledge of the filesystem, which is identified by a public key.
Giving write permissions, in contrast to some cloud storage providers, is only possible by giving that ability to individual users.

The underlying technology is the peer-to-peer filesystem hyperdrive, which already provides a robust and powerful base. However, it does not support access control. A hyperdrive instance is identified by an Ed25519 public key, which is used to sign the drive's contents. Anyone who knows this key can download the data from other peers and read all of it. The system does not support adding writers (yet) – the only way of archiving "write permissions" is mounting another hyperdrive's directory by referencing to it and handling it as a part of its own.

The peer-to-peer nature of the application limits it to the usage of native application technology, as web technology does not permit creating arbitrary TCP or UDP links. Further, for the data to be available for download, some device must share that data. In practice, that means that if a user wants to send data to someone else, both devices must be online simultaneously. For a peer to download and share the data, it is not necessary that this peer is able to read the contents – anyone that knows the public key of the drive can download, verify and share all of its data. This enables services, such as cloud storage providers, to keep the data available to others without compromising its confidentiality.

For the purpose of the master thesis the application does not feature any possibility to upload the data to a storage provider, but the internal design is done in a way it enables such an addition for future work.
An additional limitation to the application is that it does not support integration with the local filesystem. Instead it provides its own interface for file management. It should be possible to upload a file by selecting it in the OS file manager or inserting it using drag & drop. Analogous to that, files can be downloaded to the local filesystem or temporarily placed into the filesystem for directly opening it in another application.
Again, this is not part of the master thesis, but the internal design choices should enable integration into the file system and the file manager of the OS as an addition for further work.

On a technical level this leads to the following requirements:

- The core requirement is the ability for the user to control who can read and write data to its drive, fully based on a cryptographic capability access control model. This allows not having to trust the code running on remote devices or even storage providers.

- Permissions need to be revocable, in the sense that future changes are not visible to a reader and a writer is not able make any additional changes if its permissions are revoked.
- The ability to replicate data and verify its authenticity without being able to read its contents. This way the user does not have to trust a storage/cloud provider to uphold the confidentiality. Even if the provider is forced to forward the data for legal reasons, it is useless to third parties.
- It needs to be possible to use the app with multiple devices simultaneously. As hyperdrive does not support concurrent writes, each device must use its own hyperdrive instance, which are then “mounted” to the root directory of the other ones. This means the identity of a user must be independent of the hyperdrive instance.
- Allowing multiple devices per user means that it also needs to be possible to remove old devices. This is (and works) analogous to revoking ordinary permissions.
- No central authority – trusting a central authority implies that these might be able to forge identities and therefore might be able to get access to encrypted files.
  For handling identity and authenticity a simple Web-of-Trust model is applied. In order for the WoT to be effective, the information about a contact needs to be presented to the user in an understandable way, such as an icon that shows if the contact is trustworthy, next to its name or profile image.
- A strategy for handling the case of a device being stolen or hacked. In this case the device not only has to be removed, but a new identity (public key) must be generated. But a malicious actor can do that as well. One proposed solution here is utilizing the WoT for handling identity conflicts. A user then can ask one or multiple trusted persons to certify its new identity and mark the identity of the attacker as fake.
- Sparse P2P data replication – data can be requested on a block-level. This allows streaming large files to a device that does not have enough free resources to store a large file. P2P data replication also allows to arbitrarily share the network load and even enables downloading from random devices in the same network. This is already enabled by hyperdrive and only needs to be considered in the internal design.
- The app needs to have full functionality if the device has no internet connection. Therefore, changes to the data, but also the settings, must be persisted for synchronizing them at a later point in time. This means that all communication works in an asynchronous fashion. The obvious solution here is to handle all communication between users and devices by writing it to a file on the hyperdrive.
- As long as this does not turn out to be impossible or infeasible, it should be possible to open CertaCrypt filesystems in a standard hyperdrive client. Public files show up as “normal” files, encrypted data should either look like random garbage or invisible altogether.

Implementation Details

As hyperdrive is implemented in JavaScript and intended to be used with NodeJS, the platform of choice obviously is NodeJS. For simplicity reasons, the user interface will be built using Electron, which runs on the top of NodeJS, so the CertaCrypt library can run in the backend process.
Like other “missing” features, the support of mobile platforms might be added in the future. Therefore, the internal design should allow to move away from Electron without too much additional effort.

The CertaCrypt File Manager will utilize the hyperspace daemon, a general purpose background process that handles local storage and networking for any kind of hyperdrive- or hypercore-based applications. Hyperspace is still under active development and its predecessor hyperdrive-daemon is still the recommended module to use, but it will replace hyperdrive-daemon in the foreseeable future, so it is logical to build new software using the newer module, even if it is not that stable at the moment.

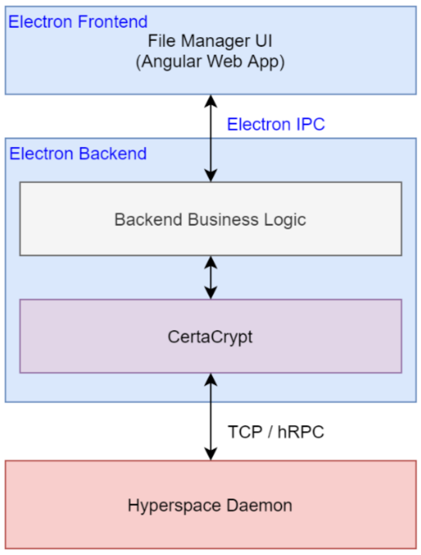

Figure 1: File Manager Architecture

CertaCrypt Core Library

Abstract Functionality

Public files are plain old hyperdrive files. Hyperdrive stores its metadata in a KV storage called hypertrie, which stores its data on a hypercore append-only log. The actual file contents are stored in a separate hypercore instance. A file's name and path only determined by key of the KV storage entry.
Public filesystem mounts are simply ".mount" files that refer to the mounted hyperdrive(s). As soon as hyperdrive supports such mounts, that functionality should be replaced.

The handling of private (encrypted) directories and files requires additional metadata. Further, also the metadata, including the filename and path, need to be encrypted. This is solved by assigning random filenames and storing them in the root directory, and by encrypting the file metadata entry. All the new and independently handled metadata are stored in the "metadata graph". Its nodes represent files, directories or "shares", which are created when read access given. They are similar to directories, but also contain additional information to provide some context to the user, but also to the underlying system. A node does not store the path of a file, only its filename. The path is derived by traversing the graph. Each link to child nodes also contains the encryption key needed to decrypt the referenced node.
Nodes refer to other nodes by storing an URL, which can either be relative, for local nodes, or absolute, when the referenced node belongs to a different hyperdrive.

It is possible that a directory refers to multiple other nodes with the same name, then it is up to the application how to handle that. An example for such a case is when write permissions to a directory are given. In that case the parent directory has two directory nodes as children that have the same name, but a different identifier and even a different storage location. A file manager will merge the two directories but will show duplicate files separately.

In contrast to some other filesystems, this graph is not necessarily a tree, as it can have multiple root nodes and can form circles. An example for an additional root node would be when a user shares a directory by URL – then a share is created, which does not have any parent nodes (from the user's drive point of view).

The capability of being able to give write permissions is implemented using the links to nodes. A link to a node can include a list of users that are have the permission to reference to other user's filesystems, therefore themselves giving write permissions. If a user does not have that permission, clients must not follow links to other hyperdrives that user refers to. A user is identified by its Curve25519 public key. In case that key changes, all permissions have to be re-written.

For the root node of the private part of the filesystem, a share is created as well ("private root node"). If the user only uses one device, the link to this share is only stored locally (not on the hyperdrive).

Each node has a unique id, which is then stored in a KV storage. It is yet to be determined if it possible to store the graph nodes in the same hypertrie as the rest of the hyperdrive metadata, without sacrificing the compatibility with other hyperdrive clients.

Communication

For the initial key exchange a special type of file is introduced. Such an "outbox" file is a public file, with its contents encrypted using a X25519 session key, which is computed using the user's private key and the recipient's public key. That file then refers to an encrypted, log-like file used for additional communication. In order to hide which user communicates with whom, these files provide no information about the recipient. Instead this follows a simple trial-and-error approach. A client has to check other user's outboxes and try to decrypt them. If that succeeds, the client knows it is the recipient. To save resources, these outbox files are deleted once the communication channel is set up.

A problem with this approach is that both parties need to know each other's public keys, as well as a hint on their intent to start communication. This can be solved by utilizing the hypercore-protocol stream-message-extensions, which allows applications to communicate over the P2P replication channel.

For doing asynchronous communication on a filesystem-basis with multiple devices, a way of handling concurrent changes to the file is required. There are many possible solutions to this – an easy solution is using automerge, a ready-to-use implementation of a conflict-free replicated datatype (CRDT) for arbitrary JSON data.

An additional problem of using log-files for communication is that hyperdrive itself does not support efficient append operations. Appends are implemented in a way that it copies the whole file and then appends the data. A straight forward solution to this is adding a file type for logs or streams, which technically is an individual hypercore instance. The module hypercore-byte-stream, which is also utilized in hyperdrive, allows easily converting the hypercore into a byte-stream.
How this can be combined with automerge could be borrowed from hypermerge.

Identity Management

Every user has to have a "/.identity" file that contains a Curve25519 public key. In addition to that, the file "/profile.json" contains the user profile, possibly referring to a public profile image.

Multiple Devices

When a user has multiple devices, technically this is not much different from two individual users sharing their whole drive. Both devices mount the other one's hyperdrive to their root directory. The only real difference is that they use the same "/.identity" file and the first device shares a "/.identitysecret" file with the other, containing the private key, effectively giving the device access to outbox files. For sharing all private files (including the "./identitysecret"), the link to the private root node is sent over the communication channel. This communication channel is initially set up by sending an URL over some other channel.

Examples

Figure 2 shows a simplified example how a filesystem looks like in practice. In that case Bob wants to share a photo of his cat with Alice. The two have already set up a communication channel using the outbox files. Alice uses two devices, which are connected in the same way as Alice and Bob are connected, but for simplicity reasons this is not included in the picture.

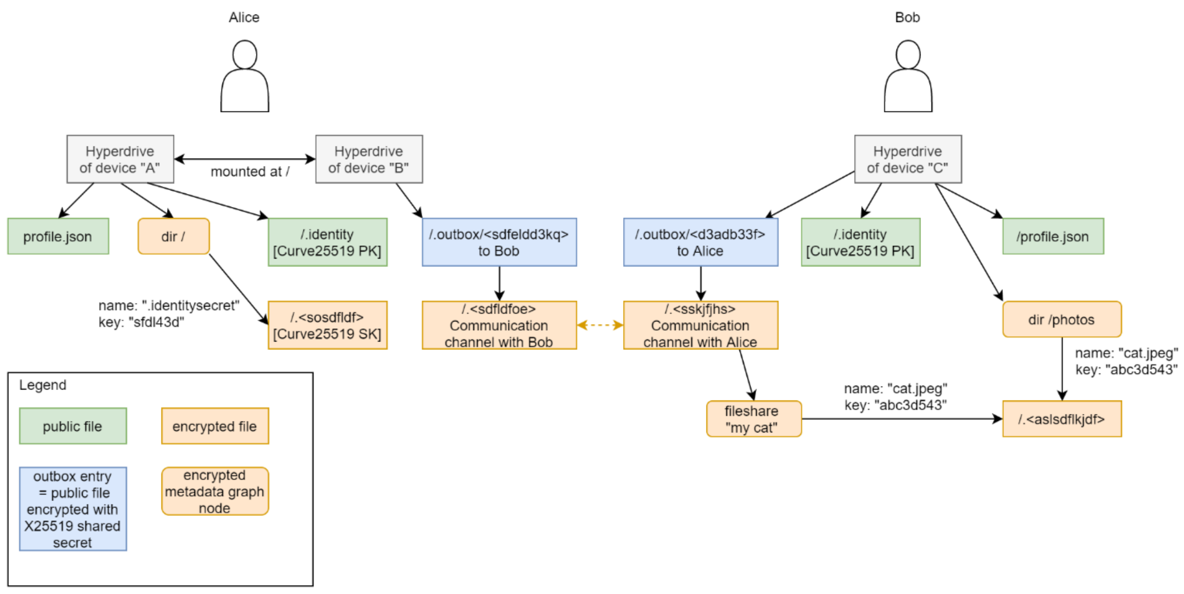
Alice
Bob
Hyperdrive of device "A"
mounted at /
Hyperdrive of device "B"
Hyperdrive of device "C"
profile.json
dir /
/.identity [Curve25519 PK]
/.outbox/<sdfeldd3kq> to Bob
/.outbox/<d3adb33f> to Alice
/.identity [Curve25519 PK]
/profile.json
name: ".identitysecret"
key: "sfdl43d"
/.<sosdfldf> [Curve25519 SK]
/.<sdfldfoe> Communication channel with Bob
/.<sskjfjhs> Communication channel with Alice
dir /photos
name: "cat.jpeg"
key: "abc3d543"
fileshare "my cat"
name: "cat.jpeg"
key: "abc3d543"
/.<aslsdflkjdf>
Legend
public file
encrypted file
outbox entry = public file encrypted with X25519 shared secret
encrypted metadata graph node

Figure 2: Simplified Example

A part of Alice's Metadata graph (once Alice includes Bob's share into her "photos" directory) is shown in Figure 3.

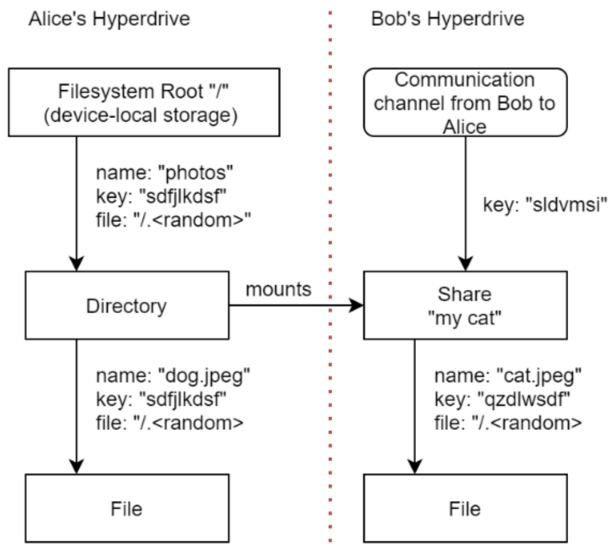

Figure 3: Metadata Graph Nodes

Architecture

To keep the code clean and easy to maintain, all functionality should be split up into smaller modules wherever possible.

In order to modify the hyperdrive functionality, some of its modules have to be modified. Luckily, the Hyper* ecosystem is designed in a way that most utilized modules can be replaced with custom implementations by injecting/passing factory methods. This way, hopefully, it is not necessary to modify the hyperdrive module at all, so it is possible to simply create a library on top that injects additional functionality into the hyperdrive module.

Components:

- Public API module
  This module combines all functionality and provides a public API. The filesystem API should be POSIX compatible (as far as hyperdrive is), but also provide additional functions for the functionality introduced by the library.
- Hypercore encryption wrapper
  Hypercore, a simple append-only-log with p2p replication capabilities, is the base component of hyperdrive. The encryption wrapper needs to be able to encrypt and decrypt the content on a per-entry basis.
  Most of the work on this module was already done for my previous work on hypercore-encrypted, what's missing is the integration of the crypto module (see below).
- HyperTrie encryption wrapper
  The hypertrie key/value store module is a core component of hyperdrive and is used to store the metadata of the filesystem. It is built on the top of a hypercore, but in order for that data structure to be functional it needs multiple hypercore log entries and therefore encryption on a hypercore level is not possible.
  As also the metadata of should be encrypted, this wrapper for the module encrypts and decrypts only the entries, heavily utilizing the Crypto module (see below).
- Corestore
  This implementation of the corestore interface wraps the hyperspace client's corestore implementation and injects the encryption-enabled hypertrie and hypercore wrappers as the default way of creating instances.
- Metadata Graph
  Handles the business logic of the metadata graph as described above.
- Outbox Module
  Handles the outbox functionality as described above.
- Hypercore-Stream filetype
  For the communication between two users/devices a filetype that supports efficient append operations is needed. This filetype can also be exposed to the public api, as it might be quite useful for other applications.
- Crypto module
  All functions that require the use of cryptography are factored out into a module that can easily be replaced. This module also takes care of the key handling, including their storage in an encrypted file. For the master thesis only a "proof of concept" implementation will be developed, but in order to get a robust level of security, this should utilize functionality of the OS, such as the use of special security co-processors.
- Cache
  Since there are a lot of calculations whose results can be re-used, the other modules should

store such for later use. This data needs to stored in an encrypted local file. Probably it would make sense to use a caching library that keeps the most frequently used values in memory.