JOHANNES KEPLER
UNIVERSITY LINZ

Author
Stefan Fixl, BSc
K01606822

Submission
Institute of
Networks and Security

Thesis Supervisor
Univ.-Prof. Dr.
René Mayrhofer

Assistant Thesis
Supervisor
Martin Schwaighofer, MSc
Dr. **Michael Roland**

May 2023

Access control for a peer-to-peer filesystem based on cryptographic capabilities

Master's Thesis

to confer the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

In this thesis we propose an access control system for the peer-to-peer filesystem Hyperdrive that utilizes a cryptographic capability system based approach. The overall goal is to simplify the development of local-first software, following the principles of prioritizing local resources instead of relying on centralized services. Hyperdrive can be a useful foundation for local-first software, but it provides only very limited access control functionality. Sharing read or write capabilities with other users is crucial for applications that enable collaborative work or social interactions. Fine-grained access control therefore is a central requirement for such use cases.

Our proposed system utilizes a graph data structure for key management and enables per-file and per-directory control of read and write permissions. Additionally, it provides a simple user system that keeps track of a user's friends and other contacts. It includes a system for initial key exchange and asynchronous communication that also works with sporadic internet access. Read and write permissions can be shared with known users, read permissions also by URL.

We implemented the proposed system as a NodeJS module and published it as an open-source library called CertaCrypt. In addition to that, we also published a demonstrator application, the CertaCrypt-Filemanager. It aims to implement an app that, from a user perspective, looks like the web interface of a cloud-storage solution similar to Dropbox or Google Drive, hiding the fact that it works completely decentralized using Peer-to-Peer (P2P) technology. This demonstrates the potential of P2P systems for implementing local-first software that replaces Software-as-a-Service (SaaS) applications.

Kurzfassung

In dieser Arbeit stellen wir ein Zugriffskontrollsystem für das Peer-to-Peer Dateisystem Hyperdrive vor, welches auf einem Cryptographic Capability System aufbaut. Unser Hauptziel ist es, die Entwicklung von Software, die lokale Ressourcen priorisiert, statt auf zentralisierte Dienste zu vertrauen, sogenannter Local-First Software, zu vereinfachen. Hyperdrive kann eine nützliche Grundlage solcher Software sein, bietet aber nur sehr begrenzte Funktionalität für die Zugriffskontrolle. Das Teilen von Lese- oder Schreibberechtigungen mit anderen Benutzern ist von entscheidender Bedeutung für Anwendungen, die kollaboratives Arbeiten oder soziale Interaktionen ermöglichen. Feingranulare Zugriffskontrolle ist daher eine zentrale Anforderung für solche Anwendungsfälle.

Unser Ansatz organisiert kryptographische Schlüssel in einem Graphen und ermöglicht das Management von Lese- und Schreibrechten pro Datei und Verzeichnis. Zusätzlich beinhaltet unsere Arbeit ein einfaches Benutzersystem, sowie ein System zur Verwaltung von Freunden und anderen Kontakten. Weiters vereinfacht es den initialen Schlüsselaustausch und inkludiert ein Protokoll für asynchrone Kommunikation, das auch bei sporadischem Internetzugang funktioniert. Lese- und Schreibberechtigungen können mit Kontakten geteilt werden, Leserrechte auch per URL.

Das Konzept wurde als NodeJS-Modul implementiert und als Open-Source-Bibliothek namens CertaCrypt veröffentlicht. Zusätzlich dazu haben wir auch eine Demonstrator-Anwendung implementiert, den CertaCrypt-Filemanager. Dieser zielt darauf ab, eine App zu implementieren, die aus der Perspektive des Benutzers wie die Weboberfläche einer Cloud-Speicherlösung wie Dropbox oder Google Drive aussieht, wobei die Tatsache verborgen bleibt, dass sie vollständig dezentralisiert mit Peer-to-Peer (P2P) Technologie arbeitet. Dies demonstriert das Potenzial von P2P-Systemen als Grundlage für Local-First Software, etwa um Software-as-a-Service (SaaS)-Anwendungen zu ersetzen.

Acknowledgements

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry of Labour and Economy, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

Contents

List of Acronyms

SaaS	Software-as-a-Service
CRDT	Conflict-free Replicated Data Types
P2P	Peer-to-Peer
DHT	Distributed Hash Table
NAT	Network Address Translation
DID	Decentralized Identifiers
ACL	Access Control List
RBAC	Role-based Access Control
PKI	Public Key Infrastructure
IPFS	InterPlanetary File System
JSON	JavaScript Object Notation
WoT	Web of Trust

Chapter 1

Introduction

1.1 Motivation

Modern Software-as-a-Service (SaaS) applications simplify our lives and allow us to work more efficiently than ever before. No wonder they gain more and more popularity as they provide seamless real-time collaboration and allow us to switch from one device to another whenever we want—even if these use a completely different hardware and software platform. Sharing our work has become as simple as sending a URL.

However, even if the utilized service is a costly premium, enterprise-grade product, we are buying that convenience also by giving up control. We are no longer in control of our data and can't be sure what that software is doing besides the things we want it to do. Typically these services also require a stable internet connection in order to work properly. Even if the colleague working on the same document is sitting next to you and shares the same WiFi, generally all the data has to be sent to some distant server where it is processed and sent back all the way to your colleague. With a decent internet connection that works quite well, but it causes a huge dependency on all hardware and software components on that path.

1.1.1 Local-First Software

This lack of control and the dependence on service providers and network connectivity is causing an opposing trend in software development—a trend back to software that works offline and towards decentralized application architectures.

Kleppmann et al. [27] coined the term *local-first software* as a paradigm for software that prioritizes local resources, such as local storage and the local network, but also enables seamless collaboration and allows working from various devices, while giving the users full control over their data and respecting their privacy. Such software ideally even works when the product or service is discontinued and the servers are shut down.

Local-first software stores all necessary data on the users' devices and does not rely on servers for the business logic. This vastly reduces the number of the all-too-common loading spinners caused by the client-server latency. Servers only fulfill a supporting role, for example by distributing data to other users or by providing additional computing resources. As a positive side-effect, this also reduces the operating costs for the server infrastructure as it reduces the overall required amount of resources.

One way of achieving complete independence from service providers and backend servers is using Peer-to-Peer (P2P) technology. Modern P2P systems provide features such as distributed databases, filesystems, messaging systems, some even blockchain-based decentralized payment and long-term storage [5, 23, 25, 42]. While coming with their own set of challenges, these already fulfill many requirements of local-first software.

1.1.2 Problem Statement

How to design and build feature-rich, reliable and resource-saving local-first software is still an ongoing topic of research [47]. As part of that, one area with open problems is how to manage access control for P2P applications in a way that does not impact the availability of private data too much, while still ensuring authenticity, confidentiality and integrity. We chose to focus on access control for a P2P filesystem library, as sharing files is required for many types of applications.

There are many modern implementations of P2P filesystems that each provide a different set of features. To narrow down the possible candidates for our access control system, we added the requirement of being a general-use open-source library. This rules out applications with a specific usecase in mind, e.g. Syncthing[1].

The P2P technology of choice, in this work, as well as for the experiments conducted by Kleppmann et al [27], is the Hypercore Protocol [17], formerly known as Dat[2], and its surrounding ecosystem of libraries and applications. Hyperdrive [SRC.16] is a filesystem built using the Hypercore Protocol. For our purposes, the best alternative to Hyperdrive we assessed is the IPFS Mutable Filesystem [24]. But it lacks a number of features Hyperdrive provides, such as file versioning and mounting other filesystems.

As the name suggests, the core module of the Hypercore Protocol is Hypercore [SRC.8], a cryptographically secured append-only log, often titled a *feed*, that can be replicated over any connection that can be mapped to a stream. Each addition is signed using an Ed25519 secret key to ensure authenticity and integrity, while the public key also serves as an address to identify the Hypercore feed. The replication can be sparse on a per-block basis and supports push and pull event handling strategies. This way additions can be propagated near real-time. The P2P networking capabilities are provided by Hyperswarm [SRC.24], which uses a Kademlia [29] based Distributed Hash Table (DHT) for peer discovery. The DHT nodes are capable of distributed holepunching to create UDP connections to peers that are behind a firewall or a router that applies Network Address Translation (NAT).

Using Hypercore as a base, Hyperdrive is a distributed filesystem that provides a POSIX-like API. It supports streaming files as they are replicated, allowing fast access to parts of very large files without having to completely download them.

As a storage and networking layer for P2P applications, the Hyperdrive filesystem with its range of useful features is a well-suited base for local-first software. As an initial evaluation, we looked at what features such Hyperdrive-based applications would likely need and what this means from a technical perspective. We found that the core requirements of these applications are various ways of sharing data with other users, such as user profiles, text messages or arbitrary files for joint work. Often also a shared space multiple users can write to is required. When using Hyperdrive for that purpose, from a filesystem point of view, this, therefore, requires controlling read and write permissions to files and directories.

[1]https://syncthing.net
[2]https://dat.foundation

A not to be neglected difficulty when using P2P technology for local-first software is achieving reliable availability without impacting privacy. If a user shares some files and does want them to be available on the network, the user either has to keep the device online and running, or has to make sure some other seeder[3] ensures its availability. In local-first software this might be another permanently running device in the local network, a generous friend or a cloud service. To keep the seeder from reading these files, a typical solution to this is to encrypt the data.

Within the community, the need for fine grained access control has been expressed frequently, but, at the time of writing this thesis, the Github issues [26, 28, 39] resulting from these discussions are still open. Some existing applications and modules aready provide simple read- and write-access control, but not on a per-file and directory level.

The Hypercore Protocol and its modules do not feature identity management and leave that to the application developer. Some applications treat Hyperdrive or Hypercore instances as equivalent to users, but a more versatile identity system could simplify the development of local-first collaborative applications a lot. There have been previous approaches for reusable identity management using Hyperdrive, dat-wot[4] and hyperidentity[5], but it seems these projects have been abandoned. We argue that a reusable identity management system with basic social features and possibly even an address book shared by applications could vastly speed up the development process.

1.2 Contribution

In this thesis, we design and implement *CertaCrypt*, a (TypeScript) framework that extends Hyperdrive with an access control system.

To get a better understanding of the requirements of applications built on top of CertaCrypt, we developed the *CertaCrypt-Filemanager*, a proof-of-concept P2P desktop application that allows to share files and directories, grant write permissions to other users and implements a simple user system. Users have a profile with a username, description and a profile picture. The ability to send friend requests, which include read permissions to the user's list of friends, also implements a simple Web of Trust (WoT) and improves the discoverability of other users. CertaCrypt features access control down to the level of single files, including the ability of revoking read and write access for future changes. This aims to implement an app that, from a user perspective, looks like the web interface of a cloud-storage solution similar to Dropbox or Google Drive, hiding the fact that it works completely decentralized using P2P technology. Both, the framework and the user interface, are open source and available on Github (see section 1.3).

In comparison to other applications in the field of open-source, P2P networked file sharing and collaborative file systems, it stands out with access control on a per-file level and the ability of revoking access to future changes. Also, it provides the possibility of commissioning a third party to ensure the availability of the data, without having to provide this third party read access.

This thesis develops and implements a *capability access control model* [13] that utilizes *cryptographic capabilities* to extend its assurances to devices and code

[3]In file-sharing network terminology, devices that provide complete datasets for download are often called *seeders*.
[4]https://github.com/jayrbolton/dat-wot
[5]https://github.com/poga/hyperidentity

out of control of the application itself. The tree-like structure of the filesystem is generalized to a graph, all access control mechanisms and the necessary additional metadata are implemented in the form of vertices and edges. Users and their properties are vertices and edges, and even internal communication is represented this way.

Read capabilities are the encryption keys required to decipher the vertices and edges, write capabilities utilize the fact that Hypercore feeds are cryptographically signed. To put it in simplified terms, write capabilities are edges to vertices stored on the other user's Hypercore feed. An exception to the capability approach are Access Control List (ACL) rules that can be attached to edges. Since it is impossible to control what data the other user writes to its Hypercore feed, these have to be checked and enforced by the reading client (also called *agent-centric security* [42]).

The graph is implemented using a specifically designed graph database that directly utilizes Hypercore for its transaction log and storage engine. Queries can be constructed using a functional approach, where the query consists of a chain of operators that, asynchronously, either yield additional results or filter for passed rules. This is heavily influenced by the Gremlin[6] query language, but is limited to read-only operations. In addition to that, applications can define views that aggregate data or influence query execution. To support more sophisticated features, edges can declare to be queried using a specific view. Most access control features are implemented that way, including the checking and enforcement of ACL rules.

CertaCrypt is meant as a middleware to simplify the development of collaborative applications that follow the local-first software principles. By using and extending Hyperdrive and the Hypercore protocol stack with access control mechanisms, this lays a foundation for collaborative P2P applications. The small amount of code and straightforward implementation needed for the *CertaCrypt-Filemanager* shows that this work largely fulfills its purpose.

1.3 Code Repositories

Our work is split into multiple libraries and applications, each distributed as a separate code repository:

- The CertaCrypt framework
  https://github.com/fsteff/certacrypt
- HyperObjects, a transaction-log based object store
  https://github.com/fsteff/hyperobjects
- The graph database HyperGraphDB
  https://github.com/fsteff/hyper-graphdb
- An extension to the graph database that implements most of the access control capabilities
  https://github.com/fsteff/certacrypt-graph
- A drop-in replaceable module that provides the cryptographic primitives
  https://github.com/fsteff/certacrypt-crypto
- CertaCrypt-Filemanager, the demonstrator application
  https://github.com/fsteff/certacrypt-filemanager

[6]https://tinkerpop.apache.org/gremlin.html

- HyperPubSub, a Hypercore protocol extension that allows peers to publish messages on specific topics or subscribe to them https://github.com/fsteff/hyperpubsub

The appendix “Project Code References” [SRC.1, ... SRC.7] contains specific Git commit hashes of the source code versions used in this thesis.

1.4 Structure

- Chapter 2 discusses how similar access control systems work and analyzes scientific literature and other related projects.
- Chapter 3 describes the inner workings of Hyperdrive and the Hypercore Protocol, also catching up to the latest versions that were developed in parallel to our work.
- Chapter 4 discusses the concepts, requirements and considerations on how to achieve the desired features.
- Chapter 5 describes the technical details on how the concepts were actually implemented. This chapter also covers the graph database that was built as a foundation for the access control system.
- Chapter 6 evaluates unsolved potential security threats and open problems. Further, it includes benchmarks for features that we expected to potentially cause performance bottlenecks.
- Chapter 7 concludes with a summary of our findings and explains possible future improvements.

Chapter 2

Related Work

Completely decentral architectures, such as the desired model of P2P based local-first software, require their access control models to have properties which are not compatible to conventional centralized access control systems. Based on the different requirements of the respective architectures, manifold approaches have been proposed and implemented.

In general, the various approaches can be categorized based on their assumptions of how trustworthy other devices in the network are. In controlled environments, such as locally running software or enterprise networks, it is typically assumed that the machine and the code enforcing the access control system can be trusted. In that case access control follows a rule-based approach that either executes requests from authorized clients or denies unauthorized ones.

In the context of local-first software that, for availability purposes, distributes data to other clients and service providers of limited trustworthiness, these other parties cannot be trusted to correctly apply access control rules. To ensure the high privacy standards local-first software promises, such software should also make sure that private data is not processed without permission.

2.1 Cryptographic Capability Systems

Wherever confidentiality is required in a possibly untrustworthy environment, cryptography is an obvious answer to that problem. Encrypting confidential data allows to decouple the storage and access to data from the capability of reading it. Digital signature algorithms further can be used to ensure authenticity and integrity, even without the writer of the data and the reader having to directly communicate with each other.

Early systems that utilize cryptography to implement a capability access control model for distributed environments have been around for quite some time. More simple cryptographic filesystems for local use even longer. A good example is CNFS, proposed and implemented by Harrington et al. (2003) [20], which separates read, write and integrity check capabilities by using an authenticated encryption scheme that uses multiple symmetric keys.

When each file is encrypted using a separate symmetric key, that results in a need for efficient key management and key distribution. A straight forward approach is encrypting these keys using asymmetric cryptography, separately per file and recipient, such that only the owner of the private key has access to it. But even if hybrid approaches to this are used, like encrypting a secret key and then using that one with symmetric cryptography, this introduces quite an overhead in performance and complexity.

A common property for capability access control models is transitive read access, meaning that read access to a directory implies read access to all of its content. For cryptographic capability systems this is easily implemented by storing the encryption keys of a directory's content as metadata attached to the directory, which in turn drastically simplifies key management.

Different approaches to key management include the use of key derivation, attribute-based encryption and proxy re-encryption schemes. In the context of access control, these are typically designed for distributed file systems or databases, such as for scalable cloud architectures and are targeted at very specific use cases. This makes it difficult, if not impossible, for these models to be applied to completely decentralized application architectures. To list a few examples, notable research in this area includes CloudHKA [12], a Bell-LaPadula based cryptographic capability security model for cloud environments, Sushmita Ruj et al.'s proposal [41] for an attribute-based encryption powered decentralized access control scheme, and Shuhua Wu et al.'s access control system for e-medicine [50] based on key derivation for implementing hierarchical user privileges.

One of the most widely adopted early software projects that uses a cryptographic capability access control model is Tahoe-LaFS [49]. It serves as a distributed data store, specifically designed as a backup system that gives the storage providers only a minimum set of privileges. It differentiates write-, read- and verify-capability keys that form a hierarchy where each *lower* capability is cryptographically derived from the *higher* one. Storage providers only need the verify-capability that allows to check a file's integrity. Depending on whether the file is an *immutable* or *mutable* file, this is either means checking a cryptographic hash or a digital signature. A read-capability consists of the verify-capability key in addition to the encryption key, which in turn can be derived from an asymmetric private key that is used as write-capability.

2.2 Blockchain-based Access Control

In the last years, the big trend in P2P software are decentralized apps that use blockchain and associated smart contract systems. Aside from payment, this can also be applied to other areas, such as for identity management using Decentralized Identifiers (DID) [19] and also for managing access control. This allows to eliminate single points of failure without having to give up techniques that otherwise would require centralized services. Blockchains can be utilized as global databases and smart contracts can be implemented to serve requests or alter data stored there.

The degree of adoption of blockchain-technology used for access control ranges from simply utilizing blockchains for storing and propagating access control rules [43, 45, 52] to smart contracts that manage multi-party authorization and proxy re-encryption schemes [3].

For the target of building local-first software, this however is only of limited use, as these access control systems do not work offline and do not prioritize local resources. While not being directly applicable, these still are noteworthy access control models and could be useful for similar decentral applications that do not need offline functionality.

2.3 Related Projects

From a conceptual point of view, the project most similar to the proposed system is Peergos[1]. It is an end-to-end encrypted, P2P file storage with an integrated social network and additional viewers and editors for various kinds of files. It is based on InterPlanetary File System (IPFS)[2] and features access control for reading and writing data, using a graph data structure for key management [22], which is based on the *cryptree* model described in [18]. It uses a tree of symmetric keys to implement transitive read access and an additional tree that manages asymmetric key pairs for write access. The verify capability is provided by IPFS, as data is identified by its hashed value. For managing user identities the developers run a central user registry for friend discovery that stores unique usernames and associated IPFS addresses, effectively implementing their own Public Key Infrastructure (PKI).

CoBox[3] is a Hypercore-based filesystem for collaboration and backup purposes. It uses a crypto-encoder[4] approach to encrypt whole Hypercore feeds with a secret key and a modified Hyperdrive implementation[5] that supports multiple writers. This allows to create shared secret filesystems, but does not give the ability to control read or write access to single files or directories.

Textile Threads DB[6] [42] is a distributed document-store database built on IPFS. It combines multiple cryptographically signed, single-writer append-only logs, similar to Hypercore, to achieve write-access capabilities using only single-writer data structures. It uses a multi-layered encryption approach for different levels of read access. Write access is achieved by specifying which logs are part of the DB, restricted by an appended ACL.

GUN DB[7] is a distributed graph database that features an access control system based on cryptographic capabilities. It is designed for web applications and communicates using WebRTC. Every user has an elliptic-curve key pair that can be used for signing data, effectively implementing a verify capability. Private data can be encrypted either by using another user's public key, or by deriving the encryption key from a secret passphrase. Write access is managed by creating certificates that contain rules specifying where the other users are allowed to write to. Conflicting parallel writes are resolved using custom Conflict-free Replicated Data Types (CRDT).

PushPin[8] is a virtual cork-board app that was developed in the course of a research project [47] on how to build P2P applications that follow the local-first software principles. It is built on Hypercore and a CRDT called Automerge to enable seamless collaboration without sacrificing the ability to work offline. It does not encrypt the data and therefore has no read access control mechanisms aside the requirement of the knowledge of the Hypercore public keys. Also, it has no mechanisms to limit what a user with write permissions is able to modify. Since each user has its own Hypercore instance to write to and PushPin does not have any mechanism to communicate requests and responses, user interactions are sent using an *Outbox* methodology that requires the recipient's client to read messages from the sender's Hypercore. Our work, from a logical view, works very similar and uses a comparable approach we also refer to as Outbox.

[1]https://peergos.org/
[2]IPFS is a content-addressed P2P data storage, see https://IPFS.io/
[3]https://cobox.cloud, https://gitlab.com/coboxcoop
[4]https://gitlab.com/coboxcoop/crypto-encoder
[5]https://gitlab.com/coboxcoop/kappa-drive
[6]https://docs.textile.io/threads/
[7]https://gun.eco/
[8]https://automerge.github.io/pushpin/

Chapter 3

Hyperdrive and the Hypercore Protocol

The *Hypercore Protocol* [17] is an umbrella term and organization for a modular P2P data network built around Hypercore. It is a collection of open-source building blocks that mostly work independently of each other, but used in combination they can play to their strengths. Modules that provide basic functionality, such as network and local filesystem access, implement specific interfaces. These modules can be drop-in replaced with others that provide the same API. The Hyperdrive P2P filesystem itself is a building block that is built on Hypercore. This means it does not come with a networking stack, and instead requires the developer to choose which one to use. Technically, that allows it to be used without any networking layer if the application does not need P2P functionality. Usually, applications that use the Hypercore Protocol building blocks utilize Hyperswarm, a DHT based networking layer that discovers and connects to peers around the globe. This modularity enables Hypercore-based building blocks to be used in the browser, but the lack of plain TCP- and UDP-based networking in browsers limits the networking capabilities required for completely decentralized P2P networks. For example, it is possible to use WebRTC[1] data channels as an alternative to Hyperswarm, but this still requires signaling servers for peer discovery and to initiate connections.

Around the year 2020, the Hypercore Protocol organization grew out of the *Dat Project* [34], a framework built for sharing large sets of data, when the release of Hypercore version 9 and other modules introduced a large set of breaking changes. The most prominent change concerned the network discovery mechanism, which was switched from a combination of DNS and the BitTorrent Mainline DHT to the specifically designed Hyperswarm DHT. Another important improvement was the use of more robust end-to-end encryption using the NOISE framework [38]. In literature Dat is often used synonymous with its most prominent usecase, the Hyperdrive filesystem [21, 40].

Our work is based on the latest stable releases during development, Hypercore v9 and Hyperswarm v2. The recently released Hypercore v10 and Hyperswarm v3/v4 add a range of useful new features that were not yet available during the research and implementation of this thesis project. For more details on that topic see section 3.5.

In December 2022 the Hypercore Protocol organization went through another rebranding. The main developers founded a new company called *Holepunch* and renamed the organization to match the new brand. Many Github repositories[2] and the documentation[3] have been moved to new websites and the previous URLs forward to the new locations. To maintain consistency in this thesis, we still use the name Hypercore Protocol.

The rapid progress of the protocol improvements made some of the referenced documentation obsolete and not every change has been documented properly.

[1]https://webrtc.org/
[2]https://github.com/holepunchto
[3]https://docs.holepunch.to

Many citations that explain Dat and Hypercore can only be seen as explanations of the fundamental principles and no longer represent an exact specification of the more recent versions. In some cases we were not able to find up-to-date documentation. To support our claims concerning these cases, the appendix “Hypercore-Protocol Code References” contains a list of references to Git code repositories that implement the stated features. These are cited as [SRC.XY].

3.1 Hypercore

As the core building block and the fundamental data structure other modules are built on, Hypercore [SRC.8] feeds are append-only logs of binary data chunks. Internally, the content is represented by signed merkle hash trees [30, 31], which allow verifying the integrity and authenticity of single entries independently of the rest of the feed [37]. A Hypercore feed can be addressed by an Ed25519 [9] public key whose private counterpart is used for signing the data, this way the knowledge of the feed also allows verifying the contents. The Hypercore module includes a mechanism for reading (or *checking-out*) a feed at a given version, with the version number being the index of the last appended block. Other modules utilize this for providing versioned data structures with very little structural overhead.

Hypercore includes its own wire protocol [36] for replicating data between peers and a file format [51] for efficiently storing the data on disk. Storage is not limited to the local filesystem. Any module that implements the random-access-storage [SRC.20] interface can be used, such as the temporary in-memory random-access-memory [SRC.21] module, or even random-access-s3 [SRC.22] for AWS S3[4] buckets.

The wire protocol can be conveyed over any binary duplex stream, but networking capabilities have to be provided by other means, e.g. by using Hyperswarm. This transport agnostic protocol is used to exchange information what Hypercore feed blocks peers can provide and which ones they need, as well as for the exchange of these blocks. At its core, the replication protocol is based on WANT, HAVE and REQUEST messages [36]. As their names suggest, these messages signal what blocks a peer is missing, can provide for download and wants to download from the recipient. WANT and HAVE messages can contain a range or number of referred blocks. REQUEST messages only specify one block at a time. Transmitted Hypercore blocks are sent using dedicated DATA messages. Peers first communicate which blocks they have and which ones they need. Only then a peer can send a request for downloading a block of data. Multiple blocks of a feed can be requested and received from different peers in parallel and, if possible, Hypercore randomly requests data from multiple peers to evenly distribute the load. When a new block is written or a peer has received a block another one has requested, this can be signaled by sending a HAVE message. This allows distributed live streaming of Hypercore feeds.

To prevent a potentially malicious peer from eavesdropping the discovery mechanism and downloading previously unknown feeds, a discovery key is used. The discovery key is derived from the public key using a cryptographic hash function [SRC.11]. A peer must know the public key when requesting blocks from another peer. To make sure this is the case, both send a cryptographic proof as part of the initial handshake [SRC.12].

The replication stream between two peers is end-to-end encrypted. With every session an ephemeral Curve25519 [7] session key pair is generated for

[4]https://aws.amazon.com/de/s3/

this purpose. This is used for the initial handshake and key exchange, Hypercore v9 uses the NOISE protocol framework [38] for a *Noise_XX_25519_-ChaChaPoly_BLAKE2b* handshake [SRC.12]. The replication stream is then encrypted using the exchanged session key and the XSalsa20 [8] cipher [SRC.12].

Hypercore version 9 has the major restriction that feeds require a linear history, which means that there must not be different versions of a block with the same index. The typical solution to make sure this never happens, is that the private key and thus the ownership of a feed is never shared between multiple devices. An application that requires multiple devices writing to the same Hypercore-based data structure should use multiple Hypercore feeds.

3.2 Hyperdrive

Hyperdrive [SRC.16] implements a filesystem on top of Hypercore. It mimics the NodeJS filesystem API[5], which is modeled on standard POSIX functions. Depending on choice of the lower-level modules for persisting and creating Hypercore feeds, the files and directories can either be synchronized with the local filesystem or stored in a more efficient format. Through Hyperspace, a Hyperdrive instance can also be mounted to the local filesystem, but only on Linux and Mac OS (FUSE). In addition to standard filesystem functions, it also provides versioning and the ability of mounting other Hyperdrive filesystems to a directory, similar to symbolic links [35].

Internally, Hyperdrive uses two Hypercore feeds, one key/value store [SRC.17] for storing the metadata and one linear-memory [SRC.19] for the actual file contents. This separation makes it possible to sparsely download files and enables fast metadata lookups. Mounting other Hyperdrive filesystems internally adds additional Hypercore feeds [SRC.18]. Unlike the structure suggests, random access writes to a file do not overwrite the linear memory in-place and instead cause the whole file to be re-written and appended to the content feed. This means editing large files is very inefficient. Solutions to this problem, such as using Linux-style inodes, have been proposed and experimented with, but at the time of writing this thesis, didn't make it into Hyperdrive [35].

The single writer and linear history limitation of Hypercore also affects Hyperdrive. Previous attempts of solving this problem replaced the metadata feed key/value store with Hyperdb [11], which combines multiple Hypercore feeds to one key/value store and resolves write conflicts using vector clocks. However, the development of Hyperdb and therefore also this approach was dropped in favor of the more flexible and low level multi-writer functionality introduced with Autobase [SRC.35], an, at the time of writing, experimental module that combines multiple causally-linked Hypercore 10 [SRC.9] feeds into one linearized append-only-log.

3.3 Hyperswarm

Hyperswarm is a P2P networking stack for connecting peers that are interested in the same topic. At its core it uses a Kademlia [29] based DHT [SRC.26], that tracks the peers for every topic. A topic is a 32-byte buffer, usually generated by applying a hash function to some data. For example, when used in combination with Hypercore, the topic is the discovery key. In addition to that, Hyperswarm applies distributed UDP holepunching techniques to connect peers

[5]https://nodejs.org/api/fs.html

that are behind a NAT or firewall [17]. When setting up a connection, Hyperswarm tries both TCP and UDP [SRC.28]. The first that succeeds is continued to be used. On top of UDP, the uTP (uTorrent Transport Protocol) [33] provides ordered and reliable delivery, congestion control and low latency. In the local network, peers can also be discovered using multicast DNS [SRC.25]. This allows Hyperswarm to be used without internet connection if both peers are in the same network.

3.4 Access Control in Hyperdrive

The one major missing feature we identified in our evaluation of Hyperdrive is fine grained access control. As previously explained, granting write access to Hyperdrive is severely limited by the linear history requirement of Hypercore. Read access is only restricted by the requirement of the knowledge of the Hypercore's public key. But for building Hyperdrive based local-first software, we want per-file control of read and write permissions. A user might want to share directories or single files and grant write access to them. We argue that this is a must-have for a universally usable P2P filesystem.

Within the community, the need for access control has been expressed frequently, but, at the time of writing this thesis, the Github issues[6] resulting from these discussions are still open. Various ways of solving this need for access control have been explored and published, but these do not provide the flexibility and the features we require.

One simple, built-in method for limiting read access to a Hyperdrive filesystem is creating multiple instances and creating symbolic links to mount them into a hierarchy of drives. Read access is then controlled by only sharing the individual public keys with those who are supposed to have read access. However, this only provides very limited flexibility and might only be suitable for certain applications.

Linking multiple instances of Hyperdrive can also be used for implementing simple write access permissions, but this limits the number of writers per directory to the one that owns the Hyperdrive instance. Some applications and modules, such as Multifeed[7] and Multi-Hyperbee[8] utilize one or more Hypercore feeds per device. This requires additional functionality for merging simultaneous writes and potential conflicts. One solution to this problem is Hypermerge[9], an application of a CRDT for JSON-like documents on top of Hypercore. But since Hyperdrive uses a different data structure, this cannot be applied here without extensive changes. A partial solution for Hyperdrive is Multi-Hyperdrive[10], which allows merging multiple Hyperdrive filesystems into one. These approaches have in common that it is only possible to grant write access to a drive as a whole, not to individual directories or files.

Achieving reliable availability for Hyperdrive-based local-first software can be challenging, especially when it comes to privacy concerns. If a user shares some files and does want them to be available on the network, the user either has to keep the device online and running, or has to make sure some other seeder ensures its availability. In local-first software this might be another permanently

[6]https://github.com/hypercore-protocol/hyperdrive/issues/190,
https://github.com/dat-ecosystem-archive/datproject-discussions/issues/80,
https://github.com/dat-ecosystem-archive/DEPs/issues/21
[7]https://github.com/kappa-db/multifeed
[8]https://github.com/tradle/multi-hyperbee
[9]https://github.com/automerge/hypermerge
[10]https://github.com/RangerMauve/multi-hyperdrive

running device in the local network, a generous friend or a paid cloud service. But a third party seeder should not be able to read the data. This means, in addition to read- and write permissions, it should be possible to grant the permission to download data without being able to read it. We categorize this *replicate capability* as part of the access control system. For more details on the requirements to our access control system see section 4.1.

3.5 New and Upcoming Features and Versions

The latest enormous step of advancements in the Hypercore Protocol, built around the protocol changes of Hypercore version 10 and Hyperswarm version 4, introduced quite a number of features that simplify access control and multi-writer data structures. Hypercore v10 had its first stable release on 2022-08-16 and other related modules followed shortly after. At the time of writing this thesis, only few higher-level modules in the ecosystem have been upgraded to support these new foundations.

The protocol improvements are hugely centered around the support of multiple writers. The most prominent new feature for Hypercore is the truncation or fork of the log, for example to be able to roll back a write operation in case of a write conflict [SRC.9]. Hypercore v10 also provides first-class support for symmetric encryption of the feed contents. By the previously introduced terminology, this implements a replicate capability when only the public key is shared with third party seeders. The new version has gone through a lot of refactoring and now has a much cleaner API that, among other things, moved from callback-based asynchronous functions to JavaScript `async` and `Promise` based processing. On the downside, this means that upgrading existing applications requires a lot of code adaptations.

The newly introduced module Autobase [SRC.35] links multiple causally-linked Hypercore instances into a single, Hypercore-like linearized view, which simplifies creating distributed, multi-writer data structures. This is achieved by embedding vector clocks into each appended block that track the latest known length of all other feeds. This solves most problems that come with parallel writes, but write conflicts are still possible. What to do in the event of a write conflict is left to the higher-level code, as this is heavily dependent on the application.

Hyperswarm had two version increments within a few months. Version 3 added NOISE protocol [38] based end-to-end transport encryption and vastly improved the holepunching capabilities [SRC.27]. Previously this end-to-end encryption was part of the replication protocol, leaving the peer discovery part in plaintext. The DHT implements security-hardening features based on s-kademlia [4] and improves the user's privacy by only sharing IP addresses after approval by the peer itself [SRC.30].

With the version 4 of Hyperswarm, for UDP-based connections, uTP has been replaced by the UDX protocol [SRC.31] that implements reliable, multiplex, and congestion controlled streams over UDP and was specifically developed for Hyperswarm.

Hyperdrive version 11 is going trough what seems to be almost a complete re-write [SRC.23]. But, at the time of writing, apart of the use of Hyperbee [SRC.34] instead of Hypertrie [SRC.17] as its metadata storage, no new features are implemented. For future work, implementing an access control system on top of this much cleaner and simpler re-write of Hyperdrive can be expected to be much easier than what had to be done for our implementation.

The development of Hypercore 10 has happened independently and parallel to this work. We expect that it will provide a very good foundation for future work in the area of local-first software.

Chapter 4

Concept

The overall goal of this work is to simplify building applications that follow the principles of *local-first software*. These principles were introduced by Kleppmann et al. [27] as a paradigm for software that prioritizes local resources, such as local storage and the local network, but also enables seamless collaboration and allows working from various devices, while giving the users full control over data and honoring privacy.

How to design and build feature-rich, reliable and resource-saving local-first software is still an ongoing topic of research [47]. This implies there is no established set of best-practice rules or guide on how to build such software. These principles require careful consideration of how to design the application architecture and what underlying technology to use.

The first consideration is how the general data structure of the access control system should look like. In terms of graph theory, filesystems can be seen as trees where files are vertices and their names are represented as labels on the edges between them. This abstraction of a directed, labeled graph data structure is easily extensible and makes it simple to apply algorithms that are typically used for graphs. Additional attributes on edges can provide semantic meaning and context. This flexibility allows us to unify almost every aspect of the access control system into one large graph. This graph is stored on a dedicated Hypercore feed, independent of the Hyperdrive filesystem. In other words, the graph is used for managing the encryption keys to the Hyperdrive files and their metadata. Our approach goes as far as that the core functionality is implemented in the form of a specialized graph database.

Another general architecture consideration that influences most other design decisions, is how clients or peers communicate changes, events and requests. An example for a cryptographic capability system would be key exchange, such as providing encryption keys for read access. If such communication between two peers is only possible though direct connections, both have to be online at the same time. But a typical user cannot be expected to keep its devices online and running all the time. To circumvent this problem, our approach is to write everything to the user's graph, including all kinds of communication. This introduces a few constraints on what type of communication is practically usable or even possible at all. For protocols that require a large number of round-trips, this introduces potentially unacceptable latency. It also means that all communication is public and therefore has to be protected appropriately. On the positive side, such a completely asynchronous architecture comes with the advantage that the latest writes to the log, and therefore also communication between two clients, can easily be relayed or temporarily stored by a third party and both clients do not need to be online at the same time. In the example of the CertaCrypt-Filemanager, for each of the user's friends, the Hypercore that stores the graph is completely downloaded and then provided to others. This vastly improves the availability of the communication data if a user has many friends.

The initial and work-in-progress concepts of CertaCrypt have been published as part of the Github repository [15] throughout their development. These concept documents helped to keep the complexity under control, for the development of the software, as well as a vehicle for explaining ideas during discussions.

4.1 Requirements

The target of modeling an application architecture for the envisioned access control system, fulfilling the principles of local-first software and providing an easy-to-use framework, comes with a long list of requirements and additional properties that would be nice to have. To formulate these requirements in a more tangible way, the following points out the core motivation behind all design decisions in the form of an example.

Imagine an application that provides the workflow and the features of a modern, cloud-based file manager, such as Dropbox or Google Drive. It allows to fluently switch between devices and collaborate with friends or colleagues, but is also handy for personal backups and provides file versioning. In case the internet connection is not reliable, it works offline and later distributes the changes once the connection works again. If a friend shares a large collection of photos and videos, it is not necessary to download all of it to view them, instead it only downloads what is required. Videos are streamed and therefore can be played immediately after the first frames are received. Read and write access to single files or entire directories can be controlled by selecting the contacts from an address book. If files are shared with strangers or persons that do not yet use this application, an invitation link can be generated.

In contrast to comparable cloud services, that application runs entirely on the user's devices. Instead of data centers as the central hub for all data flows, the communication is sent over the shortest route from one device to the other. There is no central authority that might stop providing a service or that can decide that certain files are against their policy. However, this means that the files will not be available to others in case all of the user's devices are offline. To solve that problem, commercial service providers or generous friends can be tasked to keep that data available. To make this possible everything is encrypted. That means potentially less trustworthy service providers or all too curious friends are not able to see the contents of the files or their metadata.

From a technical perspective, these objectives come with a lot of requirements. Most of them can already be solved by using Hyperdrive, but as explained in section 3.4, Hyperdrive is very limited in terms of access control. We identified the following requirements for the access control system:

- File-level control over read access by encrypting data and metadata introduces the problem of how to efficiently distribute the encryption keys. Ideally, sharing only a single key gives read access to a defined set of files. For simplicity, the access control system has a transitive read access policy where read access to a directory implies access to all of its content.
- Permissions need to be revocable, at least for future changes. This means after read permissions are revoked, future writes to the filesystem have to be encrypted with keys the affected user does not have access to. If this user also had write permissions, future writes to the shared drive must not be possible, but previous writes should not be lost.

- A shared directory has one owner and only this user must be allowed to make changes to the permissions. Support for additional administrators would be nice-to-have, but is not essential and therefore can be seen as a possible future improvement.
- Hyperdrive features built-in versioning. The access control system therefore should support this as well.

In order to fulfill the principles of local-first software and to provide a pleasant user experience, the implementation should provide additional features:

- The application has to work without internet connection. This implies that all written data needs to be communicated asynchronously. For communication between users, request-response mechanisms typical for client-server applications have to be avoided, as temporarily bad or non-existing internet connection would introduce extreme latency.
- The internal data structure should allow sparse replication, such that even for metadata, only what is necessary needs to be downloaded.
- The ability to write to a shared filesystem while being offline rules out the existence of locking mechanisms. Simultaneous writes to files and directories, as well as possibly unavoidable write conflicts need to be handled.
- Since Hyperdrive and the Hypercore Protocol do not provide a user system, at least a simple one needs to be implemented. A built-in list of contacts and friends can serve as a decentral WoT and is essential for comfortable user experience.

4.2 Capability-based Access Control

In a broader context, capability-based access control is often used for operating systems on the kernel level, such as for KeyKOS [10] and EROS [44]. There, a capability is a token a process can use to access the associated resource. In comparison to role-based access control or similar models, this allows very fine grained control and can be independent of the user the process is assigned to. In this type of capability system this is assured by the kernel.

With cryptographic capabilities, instead of having an entity that verifies a passed token, access control is assured using cryptography. For example for read access this can be implemented by encrypting the data with a symmetric cipher and the encryption key serves as the capability.

4.2.1 Replicate and Verify Capability

The knowledge of the public key of a Hypercore enables to find peers that serve its content, to download it from them and to verify its authenticity. CertaCrypt uses a separate Hypercore for its graph database, so in order to replicate all data of a user, this feed's public key has to be known in addition to the Hyperdrive public key. This capability is not bound to single files or directories, but effectively to all data of the user. If multiple users have write access to a directory, the public keys of these users are separate capabilities.

4.2.2 Read Capability

As already explained, the read capability of a file or directory is the secret key of a symmetric cipher. Our choice fell to ChaCha20 [6], a stream cipher that can be used like a block cipher in counter mode. For more details on the choice of cipher see section 5.2.

The Hyperdrive-internal metadata cannot be encrypted at Hypercore block level. To allow random access, the internal prefix tree data structure has to be in plaintext, so we only encrypt the metadata. As this would reveal filename and hierarchy, encrypted files are written to a single hidden directory in Hyperdrive, using only numeric IDs as filenames. The actual filenames and paths are stored as labels on the edges of the graph.

To simplify the key management, read access to a directory implies read access to all of its contents. This transitive read access is implemented by appending the read capabilities to edges that refer to the contained files and directories.

4.2.3 Write Capability

Direct write access to a Hypercore is only possible using its corresponding private key, which therefore can be referred to as a direct write capability. This key must not be shared between devices, so data written by another user has to be stored on this user's Hypercore. In order to permit write access to a directory, the owner has to refer to a vertex stored on the other user's Hypercore, which in turn can be modified to have edges to newly written files. In comparison to the other capabilities, this indirect write capability is a combination of the other Hypercore's private key and the fact that the feed is referred to.

4.2.4 Example

Figure 4.1 visualizes the three different types of CertaCrypt capabilities. The graph vertices are stored on the Hypercore feed in the form of a transaction-log based object store where each object/vertex has a numeric ID. The transaction information is in plain text, the vertices are each encrypted using an individual key.

With the knowledge of the Hypercore's public key, the replicate capability, it is possible to decode the transactions. The transaction information contains the mapping of vertex IDs to their corresponding block number, this way a reader holding only the replicate capability is able to decode where the graph vertices are stored in the log, but cannot read their content.

In this example, the file vertex in block 0 was written prior to the directory vertex in block 3. The directory vertex has an edge to the file. This edge not only contains the filename, but also the encryption key. The directory also refers to another directory in a separate Hypercore feed. In other words, someone else has indirect write access. When queried, these two directories are merged into one view that contains both files.

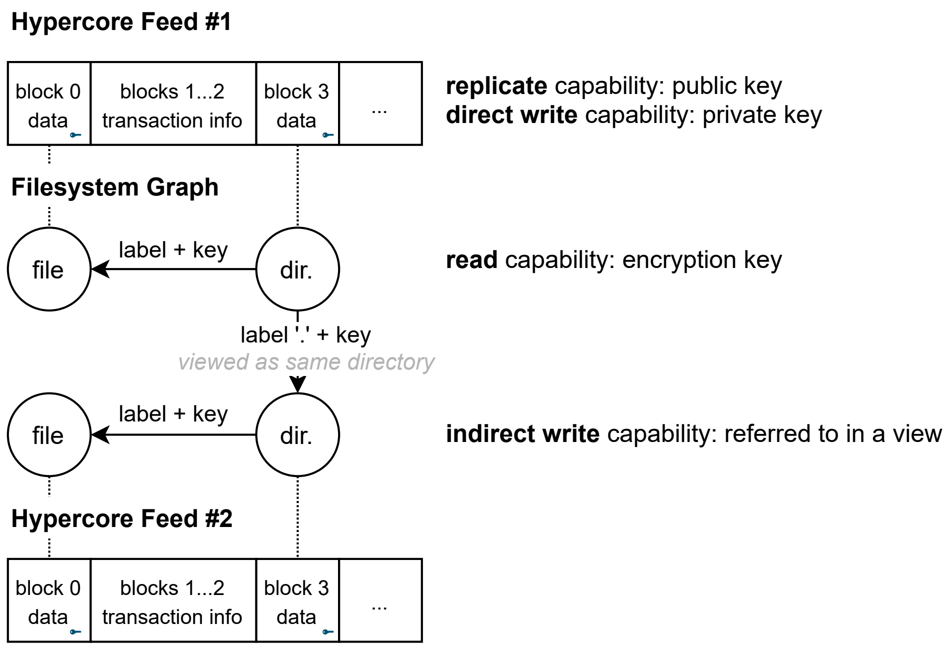

Hypercore Feed #1
block 0 data
blocks 1...2 transaction info
block 3 data
...
replicate capability: public key
direct write capability: private key
Filesystem Graph
file
label + key
dir.
read capability: encryption key
label '.' + key
viewed as same directory
file
label + key
dir.
indirect write capability: referred to in a view
Hypercore Feed #2
block 0 data
blocks 1...2 transaction info
block 3 data
...

Figure 4.1: CertaCrypt Capabilities

4.3 Threat Model

4.3.1 Actors and Components

In contrast to client-server applications, an actor in the proposed system can have multiple roles at once. Such roles can be the owner of a filesystem, a writing participant of a shared drive, the reader of shared files, a seeder of files, the contacts of a user and also a node in the DHT.

Figure 4.2 shows the major components of the CertaCrypt-Filemanager, excluding the user interface. All data structures CertaCrypt reads and writes, including the filesystem and the graph, are stored in Hypercore feeds, which serve as an important layer of abstraction. The networking capabilities and the storage layer are provided by HyperSpace, which can be run as a separate

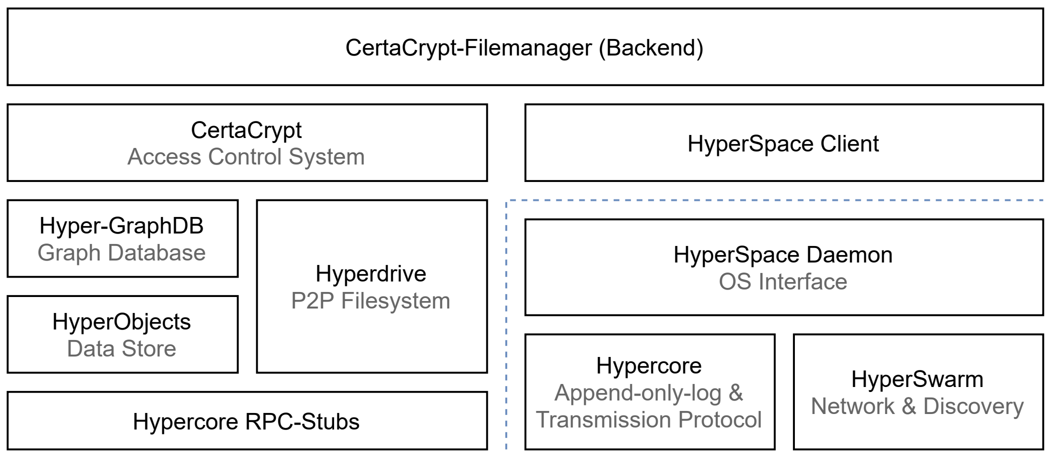


Figure 4.2: Major CertaCrypt-Filemanager (Backend) Components

daemon process. The HyperSpace RPC client only provides stubs of Hypercore feeds.

4.3.2 Assets

For a P2P filesystem the main assets are the files—both content and metadata. This also includes the file sizes, modification times, owners, writers, directory paths and access control settings. File contents and metadata should be confidential, have guaranteed integrity and need to be kept available. Especially ensuring availability can be a challenging task, as this includes the discoverability of the file in the DHT.

For some malicious actors it could also be interesting which users actually read or share files. It could even be of interest which users look up the directory structure. Another asset is a user's network of contacts and information about what these users share with each other.

4.3.3 Unauthorized Reads

Anyone who has knowledge of a CertaCrypt-protected Hyperdrive can request the data it stores from other peers. A malicious or overly curious actor could try to read files or metadata without authorization. This is prevented as all file contents, metadata and graph vertices are encrypted. Read authorization is synonymous to possessing the secret key(s) required to decrypt the data.

4.3.4 Unauthorized Writes

An attacker who disguises itself as a seeder could try to modify the distributed data by changing the content of the Hypercore feeds. This is already mitigated by Hypercore. A peer that receives data validates the signature that has to be sent as part of the Hypercore metadata. Hypercore internally creates a merkle tree, a tree of cryptographic hashes of all blocks, whose root node is signed using the Hypercore's private key.

4.3.5 Invalid Data

A main difficulty with a multi-writer data structure based on multiple Hypercore feeds, is that it is not possible to control what other users write to their feeds. Data read from another feed always has to be checked for protocol and format violations. If write access to a directory is granted by adding an edge to the other user's graph, the vertices there can contain anything. The vertices on this sub-graph can have edges to the graph of a third party, what effectively would mean that this third party also has write access to the directory. To limit what a sub-graph is allowed to contain, access control rules can be appended to any edge. When the directory contents are read, the query engine of the graph database checks if the other user's graph is compliant with these rules and removes all vertices from the result set that violate them. Typically these rules limit the sub-graph to be stored on one Hypercore only. This prevents the other user from giving write access to a third party.

4.3.6 Data out of Context and False Attribution

The unidirectional structure of the graph allows assigning existing sub-graphs to other directories. This potentially takes the contents out of context and gives them a different meaning. Such an attack could cause the victim to get into personal or legal trouble. This could be solved by adding references from child vertices to their parents, e.g. from files to their parent directories. The current implementation does not protect against such attacks, but this could be added as a future improvement.

4.3.7 Stolen Capabilities

The use of cryptographic capabilities when sharing read access also has the drawback that these capabilities could be stolen or passed to someone else, for example using social engineering techniques or malicious software on a user's device. Partially, this is mitigated by a mechanism that is required for the revocation of read permissions (see section 4.6). In order to ensure future changes cannot be read if a read permission is revoked, all files and their parent directory vertices are encrypted with new encryption keys each time they are written to. If the encryption key of a single file or directory is stolen, future changes to the affected files are not readable. Files and directories that are shared by URL need a mechanism to update the capabilities to the new encryption keys. For this purpose not the keys to the files themselves are shared, but instead special *share* vertices are created. If the shared URL is stolen or passed to a third party, that consequentially gives read access to the shared files, including future changes. In case the attack is detected, the share can be marked as revoked and does not get updated on future changes.

Write capabilities ideally never leave the Hyperspace daemon. In case a private key is stolen, the affected user is completely compromised. Write permissions to other user's filesystems can be revoked, but other shared data has to be treated as possibly malicious.

4.3.8 Denial of Service against the Network

A malicious actor might want to disrupt a running application or the availability of data. There is a huge variety of attack vectors for denial of service attacks when working with P2P software.

Hyperswarm v2 [SRC.26] utilizes a custom implementation of the Kademlia DHT, which is vulnerable to several attacks that allow modifying and disrupting the lookup of values [4]. With Hyperswarm v3 [SRC.27] a range of defense mechanisms on the network layer is introduced, based on the improved S-Kademlia [4]. This includes more robust DHT routing, node-ID assignment based on a crypto puzzle and lookup over multiple disjoint paths.

Unfortunately, the Hyperspace module [SRC.33] we use for the CertaCrypt-Filemanager only works with the older version 2 and might therefore be susceptible to denial of service and other DHT related attacks. Mitigating this would require extensive modifications to this module and our initial effort to do so[1] was dropped due to a lack of resources. This is an important task for future work.

[1]The abandoned fork can be found at https://github.com/fsteff/hyperspace.

4.3.9 Denial of Service against the Client

When a malicious actor gets a victim to read data from a Hypercore, it is almost impossible to know if the amount of data stored there is legitimate or an attempt to overload the application. This also applies to the proposed system, which contains a few relatively resource-heavy mechanisms that an attacker might exploit. The current implementation does not have any protection mechanisms against such attacks.

Another form of denial of service is when a writer in a shared directory, maliciously or unintentionally, overwrites files without providing the content to other peers. A possible solution to this would be to offer older file versions if the latest one is not available. The current implementation does not provide a dedicated API for listing and reading versions of files in shared directories. The effort for implementing this should be modest, but due to a lack of resources it is left for future work.

4.3.10 Necessary Assumptions

This thesis mostly focuses on the core aspects of the proposed access control system and relies on a large number of different components. These components are carefully selected, but a major security flaw in one dependency might compromise the whole system. In order to keep the scope of this thesis within feasible bounds, we need to make the following assumptions:

- The local device and the operating system can be trusted to work as expected and the device does not contain any malicious software. This also means that files written by the application are not modified.
- The Hypercore protocol stack is secure against remote code execution and correctly validates data integrity.
- This work, as well as the Hypercore protocol, makes heavy use of cryptographic primitives provided by libsodium. We have to assume the utilized library to provide the promised level of security.
- The utilized JavaScript modules work as expected and do not behave maliciously. According to Github[2], the CertCrypt-Filemanager depends on 331 NPM modules. Given our limited resources it would be infeasible to do an in-depth check of all these packages.

4.4 Identity and Communication

As shown in figure 4.3, every user has a set of vertices that are accessible to everyone with the knowledge of the *User URL*, which contains the user's Hypercore public key and points to a vertex. This *Public Root* vertex only serves as an entry point and has edges to the other vertices that fulfill separate functions. The *User Identity* contains a Curve25519 [7] public key that can be used for initial key exchange and is used for the *Outbox* feature. The *Profile* contains a JavaScript Object Notation (JSON) encoded user profile that contains user-defined information such as username, a short description and a URL to a profile image. The *Pre-Shared Vertices* are utilized to bootstrap write access to directories. Other users can refer to them in order to grant write access. For more details on pre-shared vertices see section 4.5.2.

[2]https://github.com/fsteff/certacrypt-filemanager/network/dependencies

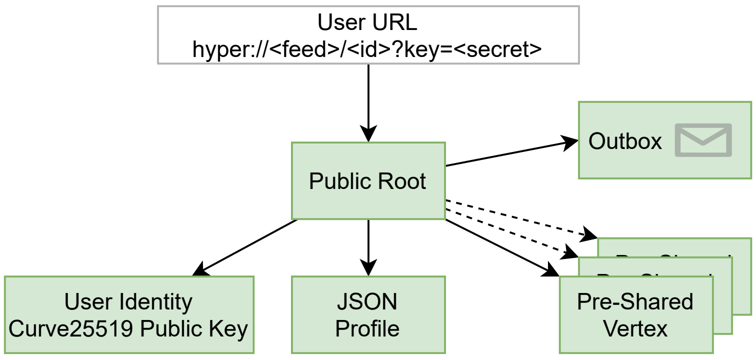


Figure 4.3: Public Vertices of a User

4.4.1 Social Graph

CertaCrypt maintains an address book for keeping track of known and trusted contacts, also called friends. From the graph database (see section 5.1.2) point of view, it is a single vertex with edges to the other users' Public Root vertices. The user can send a friend request to other users. This request contains a URL that grants read access to the address book and can be seen as a request to do the same. This aims to implement a very simple WoT and improves the discoverability of users, but also has the drawback that the shared users' profiles and identities are passed without their explicit consent. Therefore, even if that information is encrypted and only accessible to those users who have knowledge of the User URL, the profile and identity of a user have to be seen as publicly accessible.

Since all communication is stored on the graph, the recipient of a request has to be aware of the sender's Hypercore feed and periodically look for incoming messages. This can cause difficulties establishing communication between two parties that both do not know the other's identity. In that case, both need to share their User URL via a separate channel and only after both parties have manually added the other user to their address book, the communication can work in both directions. Sharing the address book with others simplifies that. If the two parties have a mutual friend, sharing the User URL on other channels is no longer necessary and communication is possible immediately. Analogously, if only one party knows the other's identity, only this one has to share its User URL over another channel.

The CertaCrypt-Filemanager adds further mechanisms for improving that situation and makes compromises in favor of the usability of the application, especially for easing the onboarding process of new users.

- If a user manually adds another user to their address book by inserting the User URL, a friend request is sent without further user interaction. This intends to increase the density and connectivity of the graph between users.
- After a friend request has been sent, the User URL of the sender is sent to the recipient using a Hypercore Protocol extension called HyperPubSub [SRC.7]. This way only one user has to know the other's identity. It utilizes existing Hypercore replication streams, as well as dedicated Hyperswarm connections, to implement a P2P PubSub system. The CertaCrypt-Filemanager subscribes to messages sent to a topic that is derived from the User Identity. The message is encrypted using libsodium's sealed box[3] method using the recipient's public key (User Identity). This stops possibly

[3]https://doc.libsodium.org/public-key_cryptography/sealed_boxes

malicious actors from eavesdropping the friend requests a user receives. When the CertaCrypt-Filemanager receives such a message containing a User URL, this user is automatically added to the list of contacts, but not to the address book. This enables communication between the two parties if only one knows the identity of the other. The downside of this approach is that this opens an attack vector for social engineering or phishing attempts, as the user cannot verify the identity of the sender of such a PubSub message. An attacker might try to impersonate someone the user knows in the real world and invite the user to a shared directory containing malicious files. As previously stated, this is a compromise taken in favor of usability of the demonstrator application.

4.4.2 Outbox

The architectural decision to write all communication to the graph comes with the challenge of how to make sure the message contents and the metadata, such as the recipient of a message, are only visible to that user and nobody else. If there was no previous communication, how can the capabilities required to read the vertices containing the messages be exchanged, and how does the recipient know where to look for messages?

The P2P social network Secure Scuttlebutt[4] writes private chat messages to the public feed, but encrypts them using the public key of the recipient and a newly generated ephemeral key pair [1]. To read the messages, others have to attempt to decrypt the data with their private key on a trial-and-error basis. A very similar approach to this can be implemented using the sealed box method libsodium provides.

Seen from the recipient's perspective, every user has an inbox that contains the messages the inboxes' owner sent to others. From the user's perspective these are outgoing messages, therefore we call it the *Outbox*. As shown in figure 4.3, the Outbox is a publicly accessible vertex referred to by the Public Root.

From the perspective of a developer using CertaCrypt, it would be logical to have a single inbox to read from, instead of having to poll other user's Outboxes. An approach to achieve this is to introduce a layer of abstraction that hides the Outboxes. The Outbox vertex has edges to the vertices that contain the initial communication attempts, but they do not specify the recipient. By utilizing a sealed box to create a secret edge to a pre-shared vertex (see section 4.5.2), the recipient is invited to write to the user's graph. This way the communication, from a logical point of view, can take place in one shared graph.

Since this single inbox would involve quite some complexity, the following simpler alternative was considered as well. Instead of referring to pre-shared vertices and thus creating a shared communication graph, each user has its own graph. The Outbox vertex works the same as for the previous approach. Its edges each contain a sealed box that contains the read capability to read the communication graph.

Both approaches have in common that, similar to private messages in Secure Scuttlebutt, users have to try to decrypt the sealed boxes to find out whether these are sent to them or if the recipient is someone else. For the first approach, additionally some sort of *Inbox request* is required, as messages can only be sent after the recipient invites the sender to its inbox. This disadvantage, paired with the additional complexity, led to a decision in favor of the simpler second approach.

[4]https://scuttlebutt.nz/

4.5 Collaboration Spaces

The local-first principles require that data can be written even if the device is completely offline. If a directory has multiple writers, that means it is possible that multiple parties write to a file simultaneously, without having any way of communicating that to each other. Logically, completely preventing such write conflicts while being offline is not achievable, at least not without severely restricting the functionality of the filesystem. Instead, CertaCrypt aims for eventual consistency [48], to achieve a consistent state for all readers at some point in time. For an extension of Hyperdrive this means that with the same information, the same state of the underlying Hypercores, all readers share an identical view on the filesystem.

Parallel, asynchronous and lock-free writes to a nested structure of files and directories can cause all sorts of write conflicts, not limited to writes to the same file. For example, if one user deletes a directory and the other adds a file to that directory, how should that conflict be resolved? Even cloud-based storage providers that offer local filesystem synchronization, such as Dropbox[5], often fall back to manual conflict resolution in case of conflicting writes they cannot automatically resolve.

We argue that a fully automatic conflict resolution strategy can only be application-specific, as complex, nested structures of interrelated files for storing application data are quite common. The design of an optimal and fully automatic conflict resolution strategy would go beyond the scope of this thesis. Instead, we introduce a very simple approach in section 4.5.1.

One possible way to solve the problem of conflicting writes would be the use of CRDTs. However, the area of CRDTs for concurrent filesystems is still an area of active research. A recent example is ElmerFS, Valliant et al. state several open problems with their current approach [46].

With the use case of the CertaCrypt-Filemanager in mind, this work simplifies the whole topic of write access to directories into what we call a *Collaboration Space*. In order to rule out a large number of edge cases, this limits the scope of supported operations and defines a conflict resolution strategy that is primarily designed for usescases where users manually share files. On the graph, a collaboration space is a special type of vertex that, by itself, does not represent a file or directory, but instead refers to a set of directories that are merged using an aggregate view when queried. Each vertex in this set represents a user with write permissions and therefore are separate sub-graphs that have to be stored on this user's Hypercore feed.

Instead of defining fine-grained permissions per file or directory using an ACL, the graph structure a user is allowed to create can be limited using a set of rules that are attached to the edges on the graph. This also allows limiting the write permission to a single file. Since the owner of the filesystem can only define these rules on the root vertex of the collaboration space, write access permissions are defined per collaboration space. From a filesystem point-of-view this is still equivalent to a directory. If a sub-directory needs different rules, a separate collaboration space can be created.

4.5.1 Conflict Resolution

The chosen conflict resolution strategy is one of the simplest deterministic algorithms we came up with and only relies on the date and time of the last mod-

[5]https://help.dropbox.com/en-us/files-folders/share/conflicted-copy

ification. If there are multiple files with the same name, the one with the latest timestamp wins. In the unlikely case of two files having the same timestamp, the order in which the users were granted write access defines the winner. When a file or directory is deleted, this file is replaced with a *tombstone* vertex, but only in the sub-graph of the writer. If a query returns a file and a tombstone with the same name and the tombstone has a larger timestamp, the file is removed from the result set.

Resolving conflicts caused by a deleted directory turned out to be more difficult than initially expected:

- If a directory is deleted and in parallel the contents of this directory are changed, we argue that these changes are most likely more important than the deletion of the directory. This conflict can be solved in various ways, for example by removing all files that are older than the tombstone. But for the sake of simplicity, our straightforward approach is to ignore the deletion of the directory if there exist changes of its contents that have a timestamp greater than the tombstone. This is achieved by propagating all changes to the parent directories and thus updating their timestamps.
- The graph DB query engine needs a built-in mechanism to reduce the set of intermediate results to be able to process tombstones for deleted directories accordingly. The concept of the graph query strategy is to asynchronously and recursively traverse the graph and only return a stream of results that match the query. In the case of collaboration spaces and in order to be able to process tombstones, the graph traversal algorithm has to assemble intermediate result sets that can be filtered if a branch contains a tombstone.

The disadvantage of this conflict resolution strategy is that it does not handle some not to be neglected edge cases. If the previous example is modified in a way that the changed file inside the deleted directory happens offline, but before the deletion, this can lead to an unexpected loss of data. This and similar problems can be solved by determining the partial order of the write events, for example by introducing vector clocks [14] or similar mechanisms. In comparison, the GUN decentralized graph DB utilizes such a conflict resolution algorithm based on timestamps and vector clocks, called the Hypothetical Amnesia Machine [32].

Attackers could deliberately cause write conflicts and set the timestamp to a future date to ensure their files cannot be overwritten. This can be mitigated by using the partial order of the write events in addition to the timestamp.

The transaction version number of the implemented graph DB, technically the Hypercore block index of the transaction marker, already implements a well suited logical clock that could be utilized for such purposes. This proposed improvement to the conflict resolution strategy would be an important feature for future work.

4.5.2 Pre-Shared Vertices

Write permissions are implemented by referring to a directory vertex that is owned by another user. In order to create an edge to this vertex, this vertex has to exist in the first place. A naive solution to this could be implemented by introducing a sort of handshake protocol where the owner first signals that write access is granted, and then the writer creates a new vertex and returns the read capabilities to the owner, who then can create an edge to it. This might work fine if, at the time of granting the write access, both parties are online

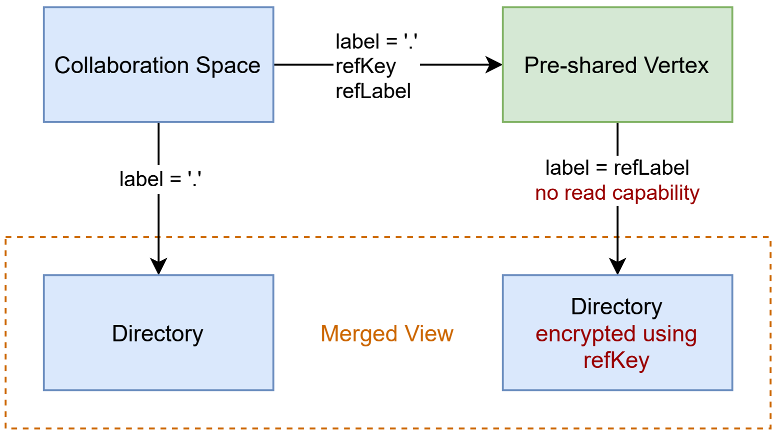


Figure 4.4: Write Permission using a Pre-shared Vertex

and can communicate with little latency. However, if this is not the case, this can cause quite some delay.

Since a more practical solution would be desirable, this is where *pre-shared vertices* come into play. The idea is that users create and share these vertices in advance, thus reducing the required number of round-trips to zero. But a user can't know how many of these vertices are required and therefore would have to provide a large enough number of such vertices to every known contact that might want to grant write access in the future. Instead, a small number of pre-shared vertices only serve as an intermediary and are not tied to a specific recipient. They are shared as part of every user's publicly accessible data, readable by everyone who knows the User URL.

When a write permission is granted, an edge from the collaboration space vertex to a pre-shared vertex is created. This edge specifies a randomly generated *referrer label* and an encryption key, the *referrer key*. With this knowledge, the recipient of the write permission creates a directory, encrypts it using the referrer key and creates an edge from the pre-shared vertex to this newly created directory. The label of this edge is the specified referrer label and since the pre-shared vertex is public, that edge must not contain the read capability. A pre-shared vertex can be used for multiple shares, therefore the randomly generated referrer label has to be long enough as to avoid accidental naming collisions. If such a collision happens, the owner of the vertex considers the second occurrence as faulty and ignores it.

Figure 4.4 visualizes this concept in a simplified manner. To ensure that the number of edges on one pre-shared vertex does not grow too large, a user can provide an arbitrary number of them, add an expiry date and potentially remove old or heavily used ones from the public list. This is not of much concern to the conceptual design, but can influence the resource usage and privacy implications. A third party eavesdropping the changes to the pre-shared vertices could gain some insights on user interaction, as the vertex IDs of the referred directories are readable and all changes to the filesystem cause these vertices to be updated. For example, a third party who has replicate capabilities for this user and for the owner of the collaboration space, could infer that the user is a participant if there are updates to pre-shared vertices in temporal proximity of updates to the collaboration space. A more privacy preserving implementation might therefore choose to create separate secret pre-shared vertices for each friend of the user.

```
rules: [{
    // prohibits any .txt files in any sub-directory and any feed
    rule: '! */**/*.txt'
    // but README.txt in the top directory is allowed
    except: {rule: '*/README.txt'}
}, {
   // restricts to this exact feed at the given version
   rule: '489894a58fad1cec3a5ba357bb4957879ff6f11a8fa56f775e4cdc8d#123/**/*'
}]
```

Listing 4.1: Example for Access Control Rules in JavaScript Notation

4.5.3 Access Control Rules

Granting write permissions to a collaboration space in itself does not imply any limitations on what the graph and therefore the directory structure can look like. Without additional rules, a user with write permissions could even refer to another user's graph, which technically gives this other user write permissions as well.

Commonly filesystems use an ACL based access control system that specifies permissions per file and directory for users, groups or dependent on the roles a user is assigned. Whenever a file is accessed, these properties are checked and if a new file is created, the permissions are set based on specific rules, e.g. inherited from the parent directory.

Applying such an ACL based access control system to collaboration spaces is difficult, mostly because the owner cannot control the content and structure of a referred graph. Given the targeted usecase of users manually sharing files, a simple, generic and efficient solution seems preferable. Since every user with write permissions has to be dealt with separately anyway, there is not much need for a group- or role-based system. In order to allow further simplifications, collaboration spaces are intended to be, permissions-wise, more-or-less homogeneous directories where every participant usually is allowed to read and write any file. Collaboration spaces can be nested. Therefore, it must not be possible to bypass limitations set in parent directories by adding rules in a child directory. These considerations suggest the use of an approach that specifies an elementary set of rules that can be extended with exceptions if necessary. Since CertaCrypt is built on top of a graph database, these rules are embedded into the graph and are already checked by the query engine.

The implemented system allows attaching limiting rules to any edge on the graph. These rules specify paths that are either allowed or disallowed, based on Unix *fnmatch*-like string pattern matching. A rule can state additional rules as exceptions that are applied as logical OR, multiple rules and rules added down the directory tree are applies as logical AND. In other words, restrictions are specified and applied in conjunctive normal form. This way edges on the path can specify additional restrictions, but cannot add exceptions that might bypass previously specified rules.

Listing 4.1 shows an example how rules can potentially look like. The first rule disallows any .txt files with the exception of a README.txt in the top directory. An exception is just another rule and itself again can state an exception. As shown with the second rule, the first element in the path describes the feed's public key and may define the version to be used. If no version is defined, or if it is set to 0, always the latest version of the sub-graph is taken. This version pinning is required for revoking write permissions (see section 4.6).

These rules are tested for every file a query returns, given its path relative to the edge the rules are attached to. For a typical collaboration space this means the rules are written to all edges that lead from the space's root vertex to the sub-graph of a user with write permissions. In such cases the referred vertices are pre-shared vertices, but these are not visible in the files' relative path and therefore have no effect on the rules.

In the case of nested collaboration spaces, the rules specified in the higher-level space cannot be overridden. Adding writers to nested spaces is therefore only possible on the owner's sub-graph, as this one has no rules attached. Theoretically, a space can be included in two unrelated collaboration spaces. This might cause inconsistent views on the data, but in terms of the access control rules, only those rules are taken into account which are written on the path the reader's query takes.

For comparison, Textile Threads DB [42] uses a technically quite similar approach for multiple writers. It also defines ACL rules the reader has to check, but in this case, these rules implement a Role-based Access Control (RBAC) pattern using a set of roles on different levels. They call this methodology, where the reader has to verify the compliance with the ACL rules, *agent-centric security*.

4.6 Permission Revocation

The fact that Hypercore is a P2P replicated append-only log implies that data once shared cannot be deleted anymore. A peer that has downloaded the Hypercore feed can always check out the graph DB with a certain version, even if later versions restrict the access to previously shared files. One method to improve that situation would be to implement a protocol extension that asks other peers to delete byte ranges in their local copy of the Hypercore feed, but a non-compliant peer might just ignore that and even proceed sharing that data with others.

Instead, CertaCrypt only supports revoking read and write access with respect to future changes. After revoking read access the read capabilities are no longer shared and when write access is revoked, the user's graph is frozen at the current version.

4.6.1 Read Access

Since old versions of the filesystem can be checked out and read anyway, it would make no sense to re-encrypt all of the files when read access is revoked. Only modified and new files need to be encrypted with a new key. This is also referred to as lazy re-encryption [22].

Key Update

Read capabilities are stored on the edges of the directory vertices, so these have to be updated and encrypted with new keys as well. Then again, the edges to these vertices have to be updated as well. This means an updated read capability for a single file requires updating every single parent directory read capability up to the filesystem root.

Read capabilities are shared with other users in the form of URLs that contain the IDs and encryption keys to dedicated Share vertices, which in turn have edges to the shared files or directories. The encryption keys attached to these edges are updated, but not the encryption keys for these vertices, else this would mean the URLs have to be shared again. This mechanism of using dedicated Share vertices for sharing read access enables revoking read access by simply marking them as revoked. A Share vertex that is marked as revoked does not receive any updates and thus the edge to the shared directory or file has an invalid encryption key once any shared file is modified.

To keep the code as simple as possible, our implementation always generates a new encryption key each time a file is modified. This has the drawback that, for every written file, all parent directories are re-written with updated encryption keys. This causes some unnecessary overhead, but otherwise additional logic would be necessary to keep track which directory has been updated already.

Referrer Key Update

Collaboration spaces that utilize pre-shared vertices for granting write access need to update their keys as well. This means the other user has to be notified that, for the next write, the referrer key is changed and the referred directory has to be encrypted with the new key. The difficulty here is that updates to the graph, as well as any communication, are not guaranteed to be available to other peers and that changes can happen while one or both parties are offline. The referrer key can be updated multiple times while the referred writer is not aware of it. Even worse, a third party with read access might not have the latest version of the owner's graph while the latest version of the referred writer is already updated. Without appropriate measures to tackle the problem, this can cause major inconsistencies concerning the referrer key and the key actually used for encrypting the referred vertex.

The desired solution to the problem of inconsistent encryption keys should work without the need of multi round-trip communication, needs to allow both parties to make changes while being offline and preferably it should nicely integrate in the previously sketched system of collaboration spaces. When the owner specifies a new referrer key, the other user should update the referenced vertex in a timely manner, but cannot be forced to do that as it might not be aware of the changed key.

Our solution to this requires a few compromises. Each time the read access to a collaboration space is revoked, the edges to the other writer's pre-shared vertices are duplicated, but with updated referrer keys. This means there can be many edges to one user, but only one is valid and a reading client has to figure that out on a trial-and-error basis. Which one that is, depends on whether the referenced user has applied the latest changes. To keep track which edges are the most recent ones and which are out-of-date, a version counter is attached to them. The old edges can be deleted only after the owner of the collaboration space observes that the requested key update has been carried out.

This solution can potentially add some performance overhead and still does not solve the problem of the edge case where a third party does not have the latest version of the owner's Hypercore feed. In this case the reader could fall back to an older version, but that only replaces one compromise with a new one. Therefore, and for the sake of simplicity, the current implementation simply throws an error if the available referrer version of the pre-shared vertex is newer than the version of collaboration space vertex. Such an exception causes the affected sub-graph to be invisible to reader, the rest of the collaboration space still works.

4.6.2 Write Access

As previously mentioned, revoking write access is as simple as pinning the referred graph to a certain version. This way, previously written files are kept and additional changes are ignored. Similar to the terminology of the Git source code version control system, a Hypercore can be *checked out* at a certain version. In the case of Hypercore, the version is determined by incrementing a counter each time a block is appended to the log. This *version pinning* is realized as an extension of the access control rules and therefore is implemented on the graph DB level. A rule can specify the Hypercore feed the files have to be stored on and this extension allows specifying a version of this feed. If the query engine encounters a rule that specifies a version, it checks out the Hypercore feed at that version.

In case a user receives write permissions again at a later point in time, this only requires removing version pinning from the access control rules. This implies that if the user made changes to its part of the graph, while actually not having write access, these changes are applied immediately. A different approach that solves this problem would be copying all the files of the user to the owner's filesystem and entirely removing the now obsolete edge. However, the decision was made in favor of the straight-forward and resource-saving approach that uses version pinning, so this is a compromise in favor of performance and simplicity.

Chapter 5

Implementation

CertaCrypt [SRC.1] is implemented as a NodeJS module and is mostly written in TypeScript, transpiled to JavaScript in the build process. The CertaCrypt-Filemanger uses this module for a desktop application built using Electron[1] and the web framework Angular[2].

5.1 Architecture

To reduce the overall complexity and to allow parts to be re-used in other projects, multiple layers of abstraction have been introduced. Figure 5.1 visualizes the modules and layers of abstraction CertaCrypt is built on. Our initial attempt for the implementation did not separate filesystem, graph and access control logic properly, and it soon became clear that the complexity of the code would become a real problem. Even with the multi-layered solution, the process of debugging a query can be quite a challenge. The lowest abstraction layer is HyperObjects [SRC.2], a transaction-log based object store on top of Hypercore. The graph database Hyper-GraphDB [SRC.3] utilizes this for storing the graph. The low-level access control mechanisms, such as encryption, are implemented as a separate module, CertaCrypt-Graph [SRC.4], which extends the graph database by overriding some core methods. CertaCrypt-Crypto [SRC.5] wraps the cryptography primitives, manages the secret keys and is intended as an easily replaceable module in case that's necessary for an application. CertaCrypt then adds the actual application logic for the access control system, provides a Hyperdrive wrapper and combines these modules into an easy-to-use library. The filemanager application additionally uses the module Hyper-PubSub [SRC.7] for improving the friend discovery process.

5.1.1 Object Store

A Hypercore feed in itself does not specify what the data stored there has to look like. It only provides methods for appending arbitrarily sized binary blocks and for reading these blocks either by block index or as a sequential stream. As a base for the graph database, HyperObjects provides very efficient random read access to objects and the ability to concurrently write objects in a way that preserves consistency even if the graph is modified at multiple locations at once.

This is achieved using a transaction log based approach, in combination with an index that contains the mapping of object-IDs to their block storage locations. Each object is appended as separate Hypercore block. Such writes to the feed can be performed by multiple transactions in parallel. When a transaction

[1]https://www.electronjs.org/
[2]https://angular.io/

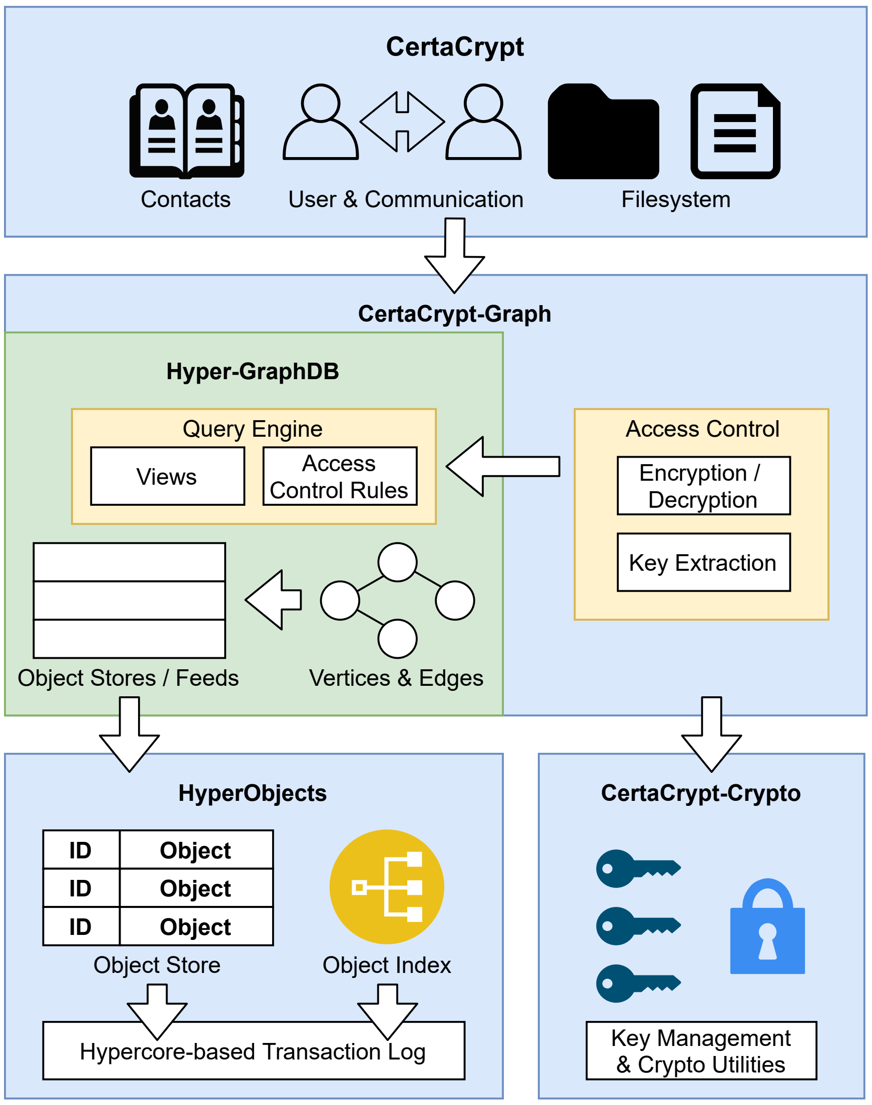


Figure 5.1: CertaCrypt Module Overview

is finished, the updates to the index are written to the feed, followed a special *transaction-marker* block. Only during this short period, the write access is restricted to this transaction.

The index has an octree structure, each node keeps track of 8 objects and additionally can refer to the locations of up to 8 other nodes. This allows partial updates of the log without having to re-write all of it. When a new object is written to the feed a unique ID is assigned to it. This object-ID is simply obtained by incrementing an internal counter, which allows it to serve as a path description when looking up the storage location in the index. The ID can be split up into triples of bits. The least significant three bits define the location within the index node, the others define the location of the index node in the tree. This octree index and the sequential object-IDs result in very efficient lookups and writes, both having an asymptotic computational complexity of $\mathcal{O}(\log n)$. For example, updating a single object with the ID of 1000 requires five blocks to be appended, the object itself, three index nodes and the transaction marker, despite the index having a size of at least 69 nodes.

Looking up an object does not require to read the transaction log from the beginning, instead the blocks are read starting at the end of the feed. The transaction marker is preceded by the root node of the index, from there the index can be searched for the object-ID. This means only those few Hypercore blocks have to be downloaded and read.

The transaction log approach also enables versioning of the objects, each transaction-marker implies a new version and points to the block that contains the previous transaction. When reading objects at a certain version, first the marker for this version has to be found. After that, the object lookup works the same as for reading the latest version.

5.1.2 Graph Database

Hyper-GraphDB allows creating and querying graphs that can consist of an arbitrary number of Hypercore feeds. It also has a mechanism for defining views that are embedded into the graph and produce aggregated or filtered results when queried.

The core element of the graph is a vertex. It stores content of a specific data type. For each data type a class has to be registered. This class needs to have methods for serialization and de-serialization to and from an arbitrary binary format. Additionally, all outgoing edges are stored as part of the vertex. These edges can contain additional metadata, such as the encryption key for the referred vertex.

Graph Queries and Views

The query language, or more precisely query API, consists of functional-programming-style chained operators that define how the graph is traversed. This is inspired by the gremlin[3] query language, but has a different syntax and is limited to read-only queries. Queries are processed asynchronously and yield results whenever they are available. This way the first results can already be processed while other parts of the graph have to be requested from other peers.

Listing 5.1 shows an example of a query that searches for vertices in the directory *photos* that match the criteria of being of type *CertaCrypt-File*. Each query

[3]https://tinkerpop.apache.org/gremlin.html

```
// get all vertices in directory photos that contain a file
let query = db.queryAtVertex(rootVertex)
    .out('photos')
    .out()
    .matches(vertex => vertex.getContent().typename === 'CertaCrypt-File')
// process the results - query.vertices() returns an async generator
for async (const vertex of query.vertices()) {
    // process result
}
```

Listing 5.1: Hyper-GraphDB query example in JavaScript

operation, such as *out*, *matches* and *repeat*, returns another query object. The operators form a chain and are then applied recursively, finally returning a number of vertices.

For more advanced usecases, queries can also return the internal state of each resulting vertex. This state contains further information, such as vertex paths, access control rules and utilized views.

Views are implemented as classes that manipulate the query logic. When the query engine encounters an edge that contains a view annotation, this view is instantiated and takes control over the sub-query. For example, write-access using pre-shared vertices is implemented using a view. This referrer-view internally evaluates the pre-shared vertex and directly returns the one that is referred to, skipping the intermediary one in the resulting state and path.

Access Control Rules

The access control rules are applied right before the query results are returned. The path that led to each result is concatenated to a string, then the rules are checked relative to their position in the graph. This verification is implemented at such a low level to make sure it is always enforced, without having to worry about incorrectly implemented views and possibly overlooked edge cases.

Read Capabilities

The CertaCrypt-Graph module adds access control functionality, such as encryption and automatic key extraction and management. Whenever a key is read, e.g. from an edge that points to an encrypted vertex, this key is put into the in-memory key store of the CertaCrypt-Crypto module. When a vertex is read from the feed, the encryption key already needs to be present in the key store. This effectively separates read access control capabilities from the query logic.

5.1.3 CertaCrypt

The main module exposes an easy-to-use API that hides most of the graph. Instead it provides classes for managing access control permissions, user profile, contacts and much more. Reading and writing files is possible through a modified implementation of Hyperdrive that is designed to be backward-compatible. A Hyperdrive protected by CertaCrypt can also contain plaintext files that can be read by the regular Hyperdrive implementation.

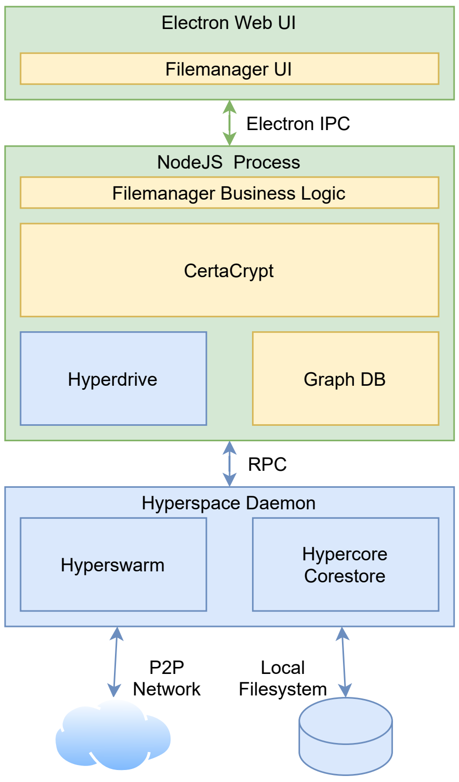


Figure 5.2: Filemanager Application Architecture

Commonly used complex graph queries are provided in the form of views. This way their results can be included in the filesystem in the form of a simple path. For example, all files and directories shared by other users are made available in the */shares* directory. That means those shared files and directories are automatically mounted to the filesystem under one common root.

The CertaCrypt module does not include networking and OS filesystem functionality, this has to be supplied by the application in the form of Corestore [SRC.32] and Random-Access-* [SRC.20] compatible plug-in modules.

5.1.4 Filemanager GUI

The demonstration application only adds a thin business-logic layer on top of the CertaCrypt module and uses Electron and Angular for the user interface. It utilizes the Hyperspace Daemon [SRC.33] as its Corestore implementation.

As shown in figure 5.2, the web-based user interface runs in a separate process that communicates with the backend logic using the Electron IPC functionality. This is important from a security perspective, as it limits the attack surface in case of cross-site-scripting and similar attacks.

5.2 Graph Encryption

Every vertex is encrypted using an individual secret key and can be updated when read access is revoked. This results in a large number of keys that are managed by the CertaCrypt-Crypto module, each identified by their vertex ID, Hypercore feed and version.

Which cipher to use for graph encryption is mostly based on the premise that the cryptography library libsodium [16] is already used by Hypercore. To minimize the number of dependencies, the selection is limited to one of the stream ciphers libsodium provides: ChaCha20, XChaCha20, Salsa20 and XSalsa20, or alternatively a higher-level hybrid method that includes authenticated encryption and generates the nonce internally.

All of these four variants of the Salsa cipher use an internal block counter in addition to the nonce, allowing them to be used similar to a block cipher in counter mode. When the cipher is used directly on the Hypercore block level, that allows the block number to be used as nonce, provided that the same encryption key is never reused for another Hypercore feed. This does not introduce any additional memory overhead and allows random access to single blocks.

Based on our threat model, we assume that the authenticity of the content of a Hypercore feed is assured by the verified signature, so we do not require a hybrid method that provides authenticated encryption.

For our purposes the 64-bit nonce of ChaCha20 or Salsa20 is enough. The extended 192-bit nonce of the XChaCha20 and XSalsa20 variants is more useful if there is no guaranteed unique source for nonce available and it instead has to be generated randomly. ChaCha20 was introduced as an improvement to Salsa20 [6]. Based on this, ChaCha20[4] with its 64-bit nonce and 64-bit internal counter was assessed to be the best method for the encryption of the filesystem graph.

5.3 Hyperdrive Encryption

The Hyperdrive-internal metadata unfortunately cannot be encrypted on the Hypercore block level. Hyperdrive internally uses a variant of a prefix tree, a rolled hash array mapped trie [2], to store the metadata. This data structure and the key strings have to be in plaintext to allow random access. Its keys, which in Hyperdrive normally contain the path and filename, are instead numeric IDs in a hidden directory. Only the content or value of the individual entries is encrypted, and in contrast to the vertices, due to API limitations, the Hypercore block number is not known in advance and thus cannot be used as nonce. In this case the choice fell to XChaCha20[5] with a random 192-bit nonce prepended to the ciphertext.

Instead of forking the Hyperdrive repository, we chose to derive our implementation from the original Hyperdrive class and override all required functions. Hyperdrive is implemented in an object-oriented fashion and utilizes a number of sub-modules and classes. As is customary for NodeJS applications, it also implements the Event Emitter pattern and uses callbacks for functions that might wait for some IO operation or event. Always keeping the ability in mind to write downwards-compatible plaintext files as well, the derived class deeply modifies the internals of Hyperdrive and still has to make sure that this does not break the original functionality. This introduced a lot of complexity and during the development often broke in completely unexpected ways. Because of these difficulties, we decided to only override the higher-level functions and skip the C-style file descriptor API Hyperdrive provides in addition. This means

[4]https://doc.libsodium.org/advanced/stream_ciphers/chacha20
[5]https://doc.libsodium.org/advanced/stream_ciphers/xchacha20

our implementation does not allow random access reading and writing, but for most applications stream-based or whole-file operations are sufficient anyway. Mounting Hyperdrives to a directory is replaced by our graph and thus also is not supported. In retrospective, forking the Hyperdrive repository and applying the changes directly likely would have been a much easier approach.

5.4 App Session

In the context of our work, the term session only vaguely relates to the way it is applied in traditional web development. Instead, it is used as an umbrella term for all user data, the way this data is stored and the business logic behind it. We call the root vertex of the user graph the session root, as access to this vertex implies read access to all of a user's data.

Figure 5.3 provides an overview over the session graph. The session root has edges to all components of the application. While these components all fulfill a specific purpose, they can have edges to each other and utilize each other's functionality. The color of the vertex indicates whether this vertex is kept secret (blue), might be shared with certain users (yellow) or is part of the public profile (green).

5.4.1 Application Data

Filesystems and other arbitrary application data can be stored under the application data vertex. This section does not need to follow any predefined structure. The CertaCrypt-Filemanager stores the user's drive root under the name *filemanager*. A user session can theoretically be used by multiple applications, but the current implementation does not feature locking and update mechanisms for concurrent writes to the graph. This could be added as a future improvement.

5.4.2 Social

The social component is referred to by three separate edges that each point to one and the same vertex. Two of these edges define an aggregate view of the data provided by the communication with other users. This component manages friends, other contacts and the internal communication. Section 5.5 describes the component in detail.

5.4.3 Shares

Files and directories shared with other users are provided using one step of indirection, the Share vertices. The re-encryption of the filesystem, enabling read access revocation, requires these shares to be updated and therefore to be centrally managed. Each share vertex contains additional information, such as the public profile URL of its owner, an arbitrary information text for debugging purposes and a flag that indicates its revocation status. The edges to share vertices can have one of two labels, either *url* or *user*, indicating whether it is shared by URL or directly with a user.

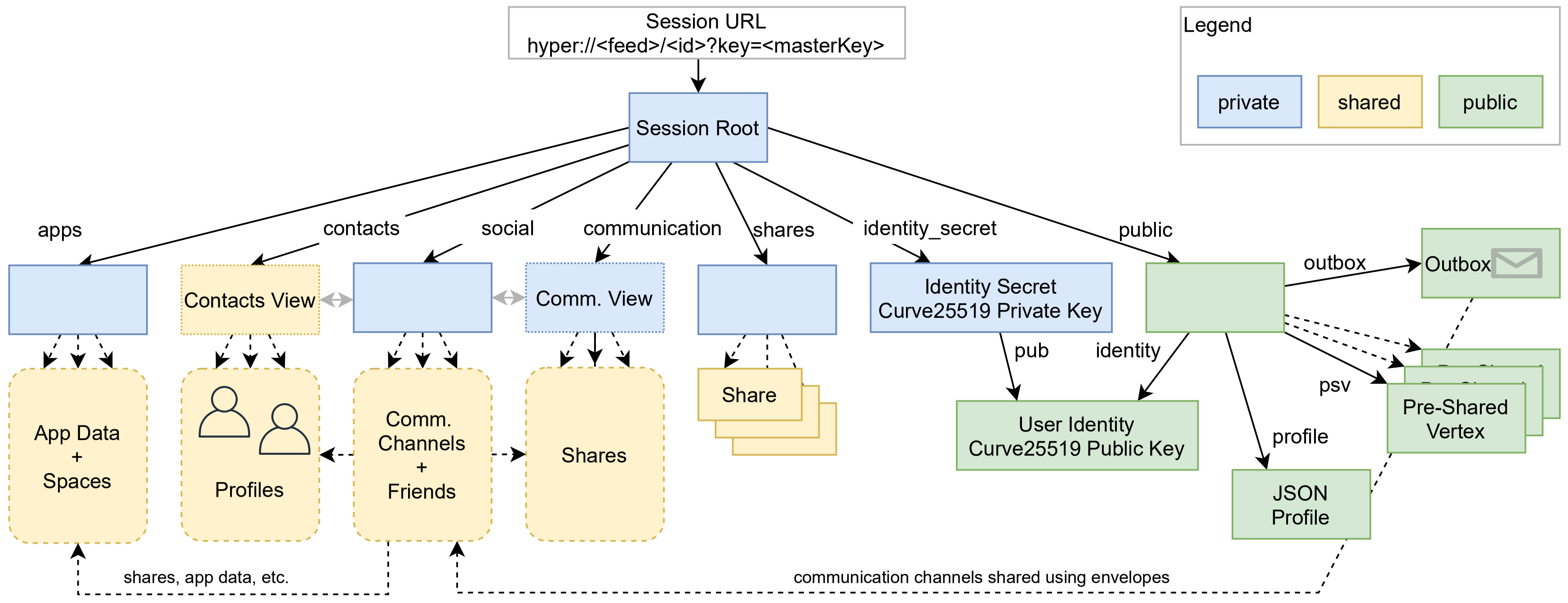


Figure 5.3: Overview over the Session Graph

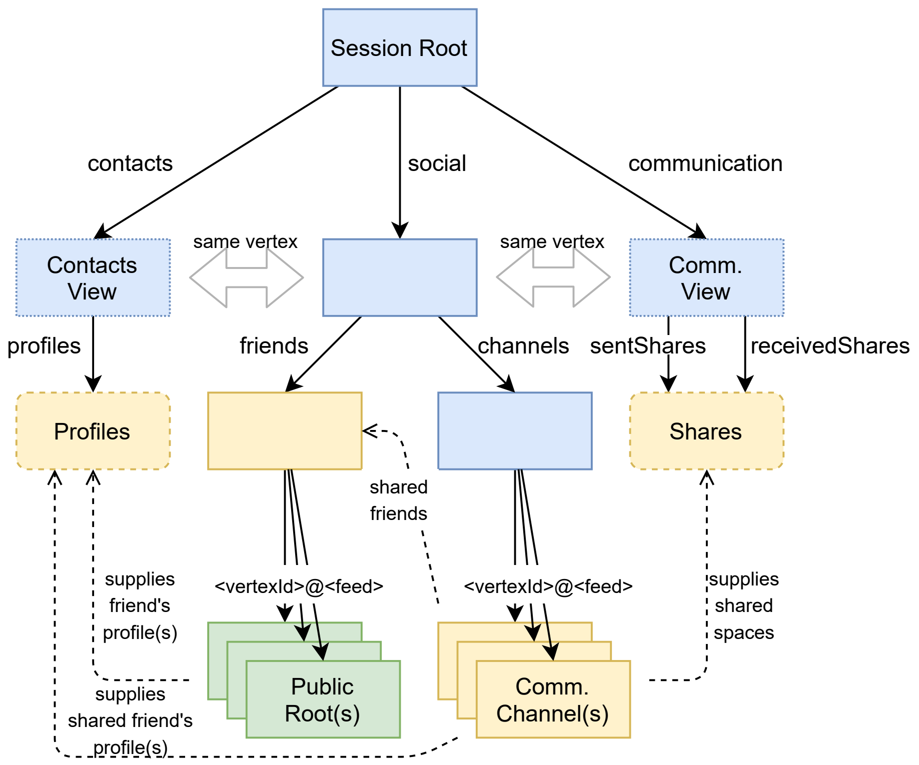


Figure 5.4: Social Sub-Graph with Views for Communication and Contacts

5.4.4 Identity and User Profile

For initiating the communication, the inbox uses libsodium's sealed box, which internally performs an X25519 elliptic-curve Diffie-Hellman key exchange [16]. For this purpose each user has a Curve25519 key pair, which is stored in two vertices, we call them User Identity and Identity Secret. The user profile is stored as a JSON-encoded string, whose contents are pre-defined, but can be extended by the application. The profile can include a URL to a profile picture stored on the user's Hyperdrive. Additionally, as explained in sections 4.4.2 and 4.5.2, the public profile contains pre-shared vertices and the Outbox.

5.5 Contacts and Communication

The social component is all about users and their interactions. It manages friends, contacts shared by other users, tracks what data is shared with others and what data others shared with the user.

When a permission is granted, the capability needs to be sent to the user. For URLs this can happen through external means, such as email, but the use of internal communication is more comfortable and secure. This is implemented by adding vertices to a part of the graph that is only accessible to the owner and the recipient, we call this a communication channel.

As shown in figure 5.4, the component also provides two views that generate aggregate data:

- The *contacts* view provides a list of profiles of all known contacts. This combines the friends and the contacts shared by other users, then fetches and decodes their JSON-encoded profiles.

- The *communication* view provides a list of sent shares and a list of received shares, implemented by querying all communication channels.

Edges to channels and friends have a distinct label, derived from the vertex IDs of the users' public root vertices and the public keys of the Hypercores they are stored on.

5.5.1 Outbox

Each time the communication channels are queried, also the outboxes of all known users are checked for new inbound messages that each may contain the capability for reading a new communication channel. Since the trial-and-error approach of the outbox is computationally heavy, previously encountered messages are cached. Therefore only new messages need to be decrypted. Technically, the outbox is part of the identity and profile component, but the caching part happens in the communication component as this concerns only the initiation of communication channels.

5.5.2 Friends and Contacts

Contacts are added by creating edges to the users' public profiles. A user becomes a friend when a friend request is added to the communication channel. This friend request contains the capability for reading the user's list of friends and additionally serves as a solicitation to do the same.

As part of the contact list, the CertaCrypt-Filemanager displays the state of the friendship, meaning whether a friend request has been sent and/or received.

5.5.3 Communication Channels

The concept of how to structure a communication channel has gone through a series of iterations. In contrast to protocols for real-time communication, e.g. for client-server application models, we only communicate in an asynchronous fashion. The protocol used by CertaCrypt only uses zero-round-trip methods. They are deliberately chosen that way to enable offline usage. Another core requirement is that it can be extended by the application, which might require requests that expect responses from the recipients.

Initially the idea was to combine the messages of the participants of a communication channel to one merged graph, or to be precise to create a view that merges the users' messages, implemented analogously to a collaboration space in the filesystem. However, this would have meant a lot of additional effort and does not provide much value, besides being a technically elegant solution. Therefore we chose to implement the simplest approach we could think of: separate communication channels that contain a vertex for each message.

A new communication channel is always started with an *Init* message, which contains additional information, such as the user URLs of sender and recipient.

Figure 5.5 shows an example for a communication channel with a friend request and one shared directory. Message types are separated by edge label. Friend requests are referred to as *requests* and shared read capabilities as *provisions*. The

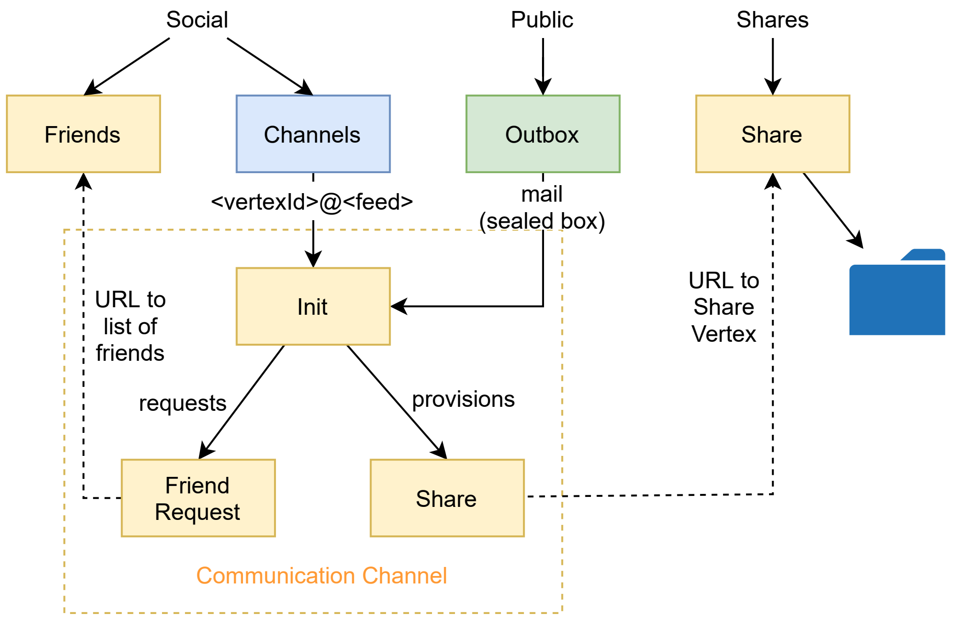


Figure 5.5: Sub-Graph of a Communication Channel

capabilities shared through these messages are implemented using URLs instead of just creating edges to these vertices. Technically, creating edges would be possible as well. It is only a design decision to separate the sub-graphs from each other.

An application can extend the communication with its own message types, which need to have a separate label. Request/response protocol extensions could be implemented by including an edge to the request when a response is created. The graph DB requires the data type of the vertex content to be known. Therefore, the message types need to be extensible as well. This is solved by encoding the message as JSON and only defining its structure in the form of TypeScript object types, which are only present in code and removed when transpiled to JavaScript. This compromise allows arbitrary messages and still is easy to work with during development.

Write capabilities, implemented as edges within the filesystem graph, are inferred and not communicated. The write permission to a directory requires the read permission. Thus the other user already knows of the directory.

5.6 CertaCrypt API

The API of the module is centered around the class `CertaCrypt`. Most of the specific functionality is implemented by additional classes, such as `User`, `Contacts` and `Shares`. These classes are managed and accessible as member variables of `CertaCrypt`. The API largely uses `async` functions, which return a `Promise` that can be accessed using the `await` keyword.

This section only covers the top level API, but a developer also has access to the internal functions if necessary. Instead of a formal definition, the function calls are shown by example to improve the readability.

```
// some corestore implementation e.g. package 'corestore' or 'hyperspace'
let corestore = new Corestore('./storage')
 await corestore.ready()
// DefaultCrypto from package @certacrypt/certacrypt-crypto
let crypto = new DefaultCrypto()
// if a session exists the session URL needs to be passed
let sessionUrl = 'hyper://...'
let certacrypt = new CertaCrypt(corestore, crypto, sessionUrl)
```

Listing 5.2: Initializing CertaCrypt

5.6.1 Initialization

Initializing CertaCrypt requires an implementation of the Corestore interface and an implementation of the interface CertaCrypt-Crypto specifies. Listing 5.2 shows an example for the initialization of the `CertaCrypt` class. If no session URL is passed to the constructor, a new session is generated.

5.6.2 Utility Functions

Building features directly on the graph database API often results in a lot of boilerplate code. CertaCrypt provides various functions that can be used to replace repetitive code:

- `let vertex: Vertex<GraphObject> = await certacrypt.path('/path-to/my-vertex')`
  It is frequently necessary to query a specific vertex by its absolute path from the session root. This function executes a Hyper-GraphDB query and returns the first result. It throws an exception if the vertex cannot be found or the result contains more than one vertex.
- `let url: string = await certacrypt.getSessionUrl()`
  This function can be used to export the session URL, e.g. for writing it to a file in the local filesystem.
- `let url: string = certacrypt.getFileUrl(fileVertex, fileName)`
  Creates a URL that points to a specific file or directory. Because filenames are specified on the edges of the graph, the names need to be included in the URLs.
- `let { vertex, name, stat, readFile } = await certacrypt.getFileByUrl(shareUrl)`
  Parses a file URL and mounts it to a temporary filesystem. It additionally returns the file metadata and a function that can be used to read the file contents. The CertaCrypt-Filemanager frequently uses this function to read the profile pictures of other users.

5.6.3 Graph Manipulation

In some cases it is necessary to directly manipulate the graph using the Hyper-GraphDB API. The currently utilized instance of the graph DB is provided as the member variable `graph` of the `CertaCrypt` class. The example in listing 5.3 creates a vertex for later use with the filesystem.

```
let appRoot = await certacrypt.path('/apps')
let driveRoot = certacrypt.graph.create<GraphObjects.Directory>()
await certacrypt.graph.put(driveRoot)
appRoot.addEdgeTo(driveRoot, 'filemanager')
await certacrypt.graph.put(appRoot)
```

Listing 5.3: Graph Manipulation

```
let driveRoot = await certacrypt.path('/apps/filemanager')
let drive = await certacrypt.drive(driveRoot)
```

Listing 5.4: Initializing the Filesystem

5.6.4 Filesystem

An instance of the Hyperdrive wrapper can be created using the function `drive`, which takes a vertex as argument for the filesystem root. Listing 5.4 continues the example from listing 5.3 by initializing the filesystem on the created vertex.

The CertaCrypt filesystem API overrides the one of Hyperdrive, but is limited to the following functions: `createReadStream`, `createWriteStream`, `mkdir`, `readFile`, `writeFile`, `unlink`, `readdir`, `stat`, `lstat`, `access` and `exists`. Similar to the NodeJS filesystem API, Hyperdrive provides an asynchronous callback-based API and a `Promise` API. Both are supported.

The filesystem API has two modes: the *encrypted* mode enables the CertaCrypt access control and the the original Hyperdrive *plaintext* mode provides downwards compatibility. Most functions of the Hyperdrive API have an *options* parameter for specifying additional settings, e.g. which encoding to use and the detail level of the result. CertaCrypt extends the options parameter to specify whether the encryption mode should be used. It can be activated by setting the `db.encrypted` flag to true. The CertaCrypt API adds the options parameter for those Hyperdrive functions that do not have it. The plaintext mode does not use the CertaCrypt filesystem graph. Therefore, the directory structures of the encrypted and the plaintext mode are independent of each other. Hyperdrive methods that are not supported by CertaCrypt ignore the `db.encrypted` flag and therefore always use the plaintext mode.

Listing 5.5 shows some exemplary usage of the promise API. The only difference to the callback API is that, instead of returning a promise, each function has an additional callback parameter.

CertaCrypt provides a database view that lists all received shares. Each share is identified by a unique, auto generated label and can be accessed by querying that path. Listing 5.6 shows how to use this view by mounting it to a directory. In this context, mounting a share to a directory means creating an edge from the directory vertex to the referred one. The `DriveShares` class provides internal functionality that allows the extended Hyperdrive to access the view. Because of this, the method `mountAt` requires passing an instance of the filesystem the shares are mounted to.

5.6.5 User and Profile

The `User` class manages the user profile, the Outbox and the pre-shared vertices. Outbox and pre-shared vertices are handled automatically. There are two ways

```
// for encrypted files and directories the 'db.encrypted' flag is required
await drive.promises.mkdir('/', {db: {encrypted: true}})
await drive.promises.mkdir('/shares', {db: {encrypted: true}})
let files = await drive.promises.readdir('/', {db: {encrypted: true}, includeStats
    : true})

await drive.promises.writeFile('/readme.txt', 'Hello world!', {db: {encrypted:
    true}, encoding: 'utf-8'})
let content = drive.promises.readFile('/readme.txt', {db: {encrypted: true},
    encoding: 'utf-8'})
```

Listing 5.5: File Access Examples

```
// mount all received shares to the /shares directory
let shares = await certacrypt.driveShares
let driveRoot = await certacrypt.path('/apps/filemanager')
// mounts the DriveShares to the Directory 'shares' of the given filesystem/drive
await shares.mountAt(drive, driveRoot, 'shares')
```

Listing 5.6: Mounting all received Shares

of getting an instance of the class:

- `let user: User = await certacrypt.user`
  The current user can be accessed as a member variable of the main class.
- `let user: User = await certacrypt.getUserByUrl(userUrl)`
  Loads another user by parsing the URL and fetching the referred graph vertices.

The following high-level functions can be used for managing and accessing users:

- `let url: string = user.getPublicUrl()`
  Generates a URL that points the the user's public root vertex.
- `let profile: UserProfile = await user.getProfile()`
  Parses and returns the user profile.
- `await user.setProfile(profile)`
  Updates the user profile.

5.6.6 Shares

Internally, the shares are managed by the `Shares` class, but the high-level API is provided by the main class:

- `let share: Vertex<ShareGraphObject> = await certacrypt.createShare(vertex, false)`
  Creates a new share that points to the passed vertex. If there already is a share for the given vertex, it can be used by setting the second parameter to true. The CertaCrypt-Filemanager reuses existing shares when a file is shared by URL and always creates a new one if shared with a specific user.
- `await certacrypt.sendShare(shareVertex, receipients)`
  Uses the internal communication to share a file with the given users.

- `await certacrypt.mountShare(targetVertex, label, shareUrl)`
  Parses the passed URL and attaches the referred vertex to the given target vertex. The label parameter specifies the label of the new edge. In contrast to `DriveShares.mountAt`, this method mounts a specific share using a defined label. It can be used to access files which are shared by URL.

5.6.7 Contacts

The `Contacts` class manages friends and contacts. An instance of the class is provided by the `contacts` member of the `CertaCrypt` class. In addition to managing friends and contacts, it provides functions for managing sent and received shares:

- `await contacts.addFriend(user)`
  Adds a user to the friend list and sends a friend request.
- `await contacts.removeFriend(user)`
  Removes a user from the friend list.
- `let state: FriendState = await contacts.getFriendState(user)`
  Evaluates whether a friend request has been sent or received. The return value is one of `NONE`, `REQUEST_SENT`, `REQUEST_RECEIVED` and `FRIENDS`.
- `let friends: User[]  = await contacts.getFriends()`
  Returns all users on the friend list.
- `let allContacts: ContactProfile[] = await contacts.getAllContacts()`
  Returns the profiles and user URLs of all known contacts by combining the list of the user's friends with each friend's list of friends.
- `let shares: CommShare[] = await contacts.getAllReceivedShares()`
  Generates an overview of all shares received from all known contacts.
- `let shares: CommShare[] = await contacts.getAllSentShares()`
  Lists all shares sent to other users.

5.6.8 Collaboration Space

Write access to directories can be managed using the `CollaborationSpace` class.

A directory can be converted to a collaboration space by calling
`let space = await certacrypt.convertToCollaborationSpace(absolutePath)`

Existing collaboration spaces can be loaded using
`let space = await certacrypt.getSpaceForPath(absolutePath)`

The class provides functions for managing write access and additionally provides utility functions for easy and efficient access to the current set of permissions:

- `await space.addWriter(user, restrictions)`
  Grants write access to a collaboration space. The second parameter is optional and allows specifying access control rules. The default rule limits the sub-graph to the user's feed. If the user previously had access permissions that were revoked, the version pinning is removed.
- `await space.revokeWriter(user)`
  Revokes the write access permission of a user by pinning the feed to the current version.

- `let hasAccess: boolean = await space.userHasWriteAccess(user)`
  Evaluates whether a user has write permissions to the collaboration space.
- `let writers: User[] = await space.getWriters()`
  Returns a list of all users with write permissions, including the owner.
- `let writersUrls: string[] = await space.getWriterUrls()`
  Returns a list of URLs to all users with write permissions. This does not require loading the users, making it faster.
- `let owner: User = await space.getOwner()`
  Loads the user that owns the collaboration space.
- `let ownerUrl: string = await space.getOwnerUrl()`
  Returns the user URL of the owner.

5.7 Filemanager User Experience

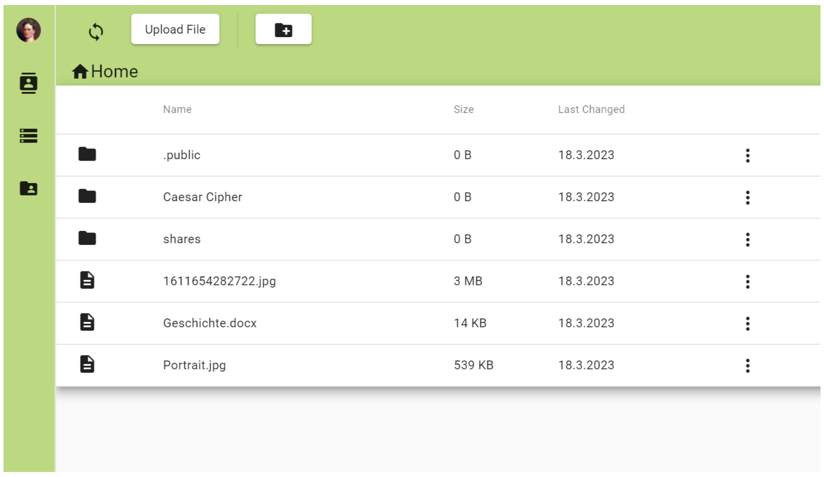
The user interface is a minimalist demonstrator for the capabilities of CertaCrypt. It uses the angular material UI component library[6]. The UI provides two views: a filesystem page and an overview of all shares. It also makes heavy use of dialogue boxes.

The left-hand menu bar contains four icons: a profile picture, a contacts icon, a drive icon, and a shared-directory icon. The profile picture leads to the user's profile dialogue, where users can edit their personal information and their profile pictures. Clicking the "share" button on the profile dialogue copies the user URL to the clipboard. Figure 5.7a shows a screenshot of this dialogue. The contacts icon on the menu bar leads to the contacts dialogue, which lists all contacts and provides buttons for sending friend requests. New contacts can be added by pasting a user URL to the input field in the contacts dialogue. This dialogue is shown in figure 5.7b. The drive icon leads to the filesystem page and the shared-directory icon leads to the share overview.

The filesystem page (as shown in figure 5.6a) is an explorer-like view that allows users to navigate through their files and directories. It is designed to resemble a typical file system view, where users can browse through their folders and files. The top menu bar has buttons buttons for uploading files, creating directories and refreshing the view. The menu bar also contains a breadcrumb navigation that shows the path to the currently viewed directory and allows to navigate to parent directories by clicking on them. The filesystem page includes a context menu that appears when users click on the three dots next to a file or folder. The context menu provides three options for managing the selected file or folder: download, share, and delete. Clicking on "share" opens the share dialogue. This dialogue lets the user share a file either by URL or by selecting the recipient from a list of contacts. Additionally, it provides a button to grant write access to a contact and the option to revoke the permission by clicking the button another time. Figure 5.6b shows the dialogue for sharing a file with contacts and figure 5.6c shows sharing by URL.

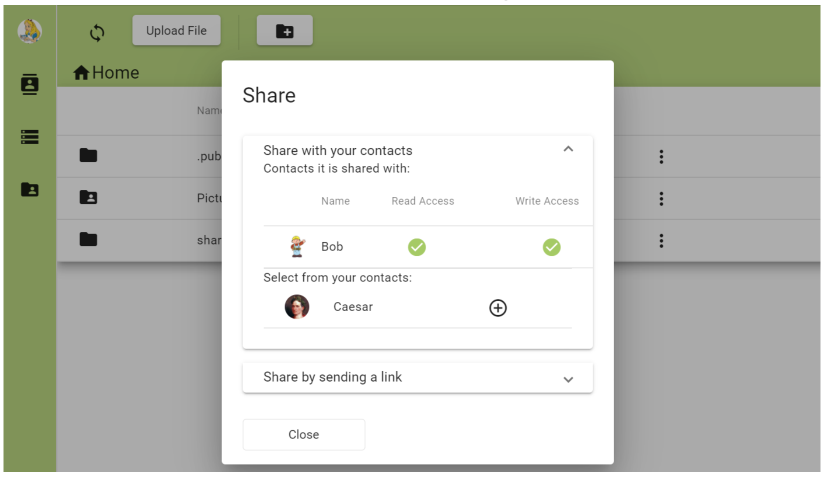
The share overview page (as shown in figure 5.6b) lists all shared and received directories and files. Additionally, it provides an input form for mounting files or directories shared by URL. The list of shared directories and files is separated into two categories: shared by the current user and shared by others. Each element includes information about the file or directory, whom it is shared with and who shared it. Users can navigate to the shared files by clicking on the paths.

[6]https://material.angular.io/



	Name	Size	Last Changed
	.public	0 B	18.3.2023
	Caesar Cipher	0 B	18.3.2023
	shares	0 B	18.3.2023
	1611654282722.jpg	3 MB	18.3.2023
	Geschichte.docx	14 KB	18.3.2023
	Portrait.jpg	539 KB	18.3.2023

(a) Filesystem Page



Share

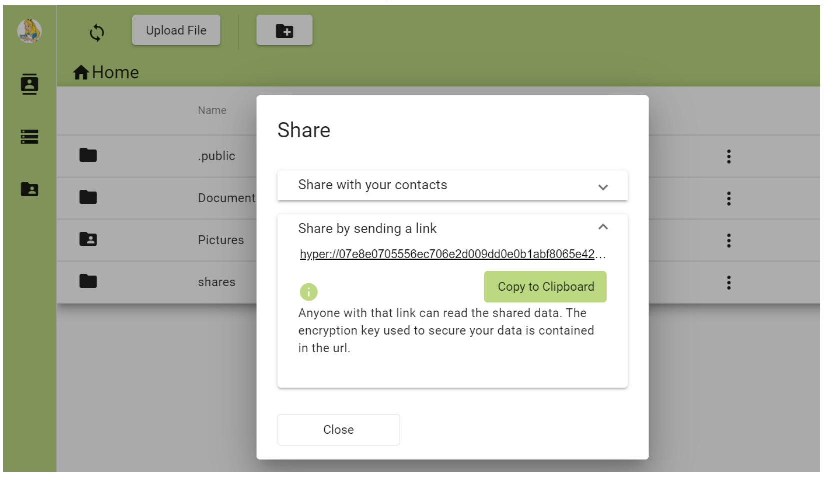
Share with your contacts
Contacts it is shared with:

Name	Read Access	Write Access
Bob	✓	✓

Select from your contacts:
Caesar

Share by sending a link

(b) Sharing with Contacts



Share

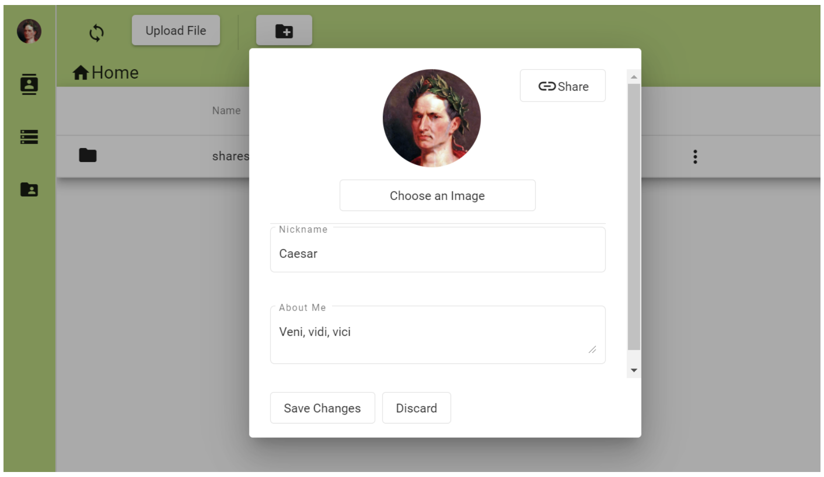
Share with your contacts

Share by sending a link
hyper://07e8e0705556ec706e2d009dd0e0b1abf8065e42...

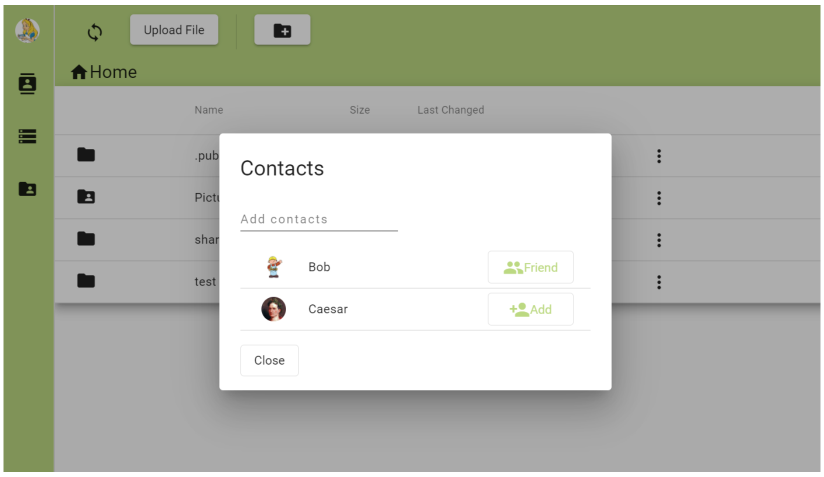
Anyone with that link can read the shared data. The encryption key used to secure your data is contained in the url.

(c) Sharing by URL

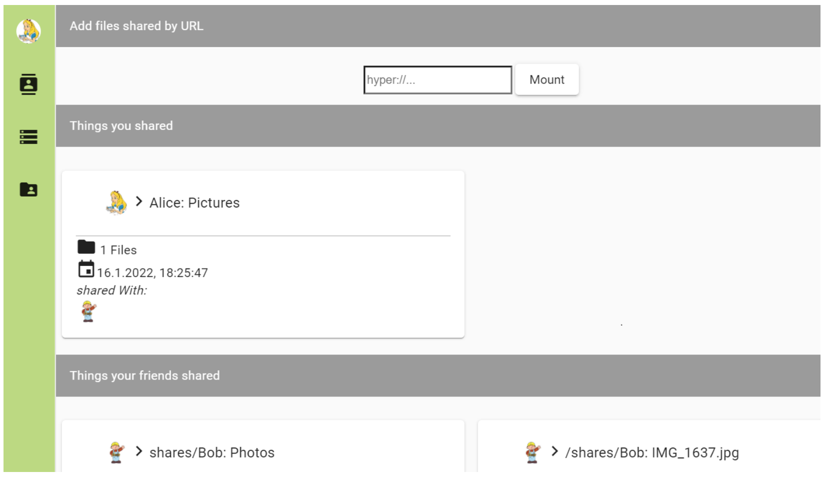
Figure 5.6: Filesystem Interface Screenshots



(a) User Profile



(b) List of Contacts



(c) Share Overview Page

Figure 5.7: User and Contacts Interface Screenshots

Chapter 6

Evaluation

6.1 Security Considerations

While CertaCrypt encrypts files, directories and their metadata, the most basic data structures are stored in plaintext. An attacker who knows the public key of a user might exploit that to gain insights. We identified multiple potential attack vectors of this category.

6.1.1 Inferring the Directory Structure

Every time a file or directory is modified, it is encrypted with a new encryption key. The new key has to be propagated to the parent directory, which in turn is encrypted with a new key as well. This lazy re-encryption enables the revocation of read access permissions, but causes an easily recognizable pattern in the transaction log. The vertices are persisted in the order they are modified, all in the same transaction. The vertices themselves are encrypted, but their IDs are not. Given there are many different write operations to analyze, an attacker might be able to infer the directory structure of the filesystem.

This potential attack vector could easily be mitigated by randomizing the order the vertices are written within a transaction. Additionally, other paths in the filesystem tree could be re-encrypted as part of the same transaction to impede an attempt of such an attack.

6.1.2 File Size

The internal data structure of the Hyperdrive is stored in plaintext. Encrypted files are stored in a hidden directory, using a numeric filename, but the storage location in the Hypercore feed can be read by anyone. Knowing the size of a file allows an educated guess of some file types. Especially if the filesystem structure is known, a directory e.g. might be recognizable as a collection of music or videos.

6.1.3 User Relations and Identity

To analyze which users interact with each other, an attacker only needs to connect to the peers in the swarm and track which peer provides which Hypercore feed ranges. A peer that provides a user's Hypercore feed as a whole, likely is either the user or one of their friends. Listening to new blocks on the feed, to a certain degree, allows identifying the owner of the feed.

6.2 Memory Usage

The fact that all version history kept available on the append-only log, raises concerns about the memory consumption and required disk space. Hypercore does not keep the whole feed loaded in memory, but it caches frequently used blocks [SRC.8]. Therefore, the RAM usage regarding the log size hits an upper limit once the maximum cache size is reached. However, the required disk space grows with each change to the graph. When the complete feeds of all of the user's friends are downloaded, this also becomes a network issue.

It is difficult to provide exact numbers for the required disk space, because there are many variables that influence number of bytes written to the feed. In general, the required space per change increases with the number of vertices in the graph. In other words, the larger the number of files in the filesystem, the larger the overhead of each write operation. This has various reasons:

- The more outgoing edges a vertex has, the larger its size. Outgoing edges are stored as part of the vertex and are persisted that way. Adding a file to a directory requires re-writing the directory vertex, including all of its edges.
- The overhead of the HyperObjects block index grows with the number of vertices it contains.
- The re-encryption of parent directory vertices up to the filesystem root means all of them need to be persisted. The deeper the folder structure, the larger the overhead.

An endlessly growing log is not suitable for a production-grade application. One solution to this problem would be pruning the transaction log. This would mean deleting all old versions of files and their metadata, but for many types of applications versioning is not required anyway. It would also be possible to keep important versions and delete the rest. The transaction log pruning could easily be implemented on the HyperObjects level. This way it would not interfere with the encryption.

6.3 Benchmarks

All benchmarks were executed on a laptop with an Intel Core i5-1035G4 CPU running Windows 11. The utilized NodeJS runtime had version 14.6.0. To ensure the code was optimized by the JavaScript runtime, a warm-up period was executed prior to each benchmark. Instead of the Hyperspace daemon, the default Hypercore Corestore [SRC.32] was used with the module random-access-memory [SRC.21] as a virtual in-memory persistence layer. Since measuring time in the µs range can be very inaccurate on normal PCs, each measurement was done on many repetitions of the evaluated operation. The code for the benchmarks and the raw results are published in the CertaCrypt Github repository [SRC.1]. Time measurements were performed using the NodeJS function `process.hrtime`, which returns the current high-resolution real time in seconds and nanoseconds.

6.3.1 Access Control Rules

The checking of access control rules requires validating the relative file path against a rule specified in a string syntax. This benchmark measured the performance hit and runtime complexity of the rule validation. For each measurement N rules were added to a collaboration space with one writer and a single

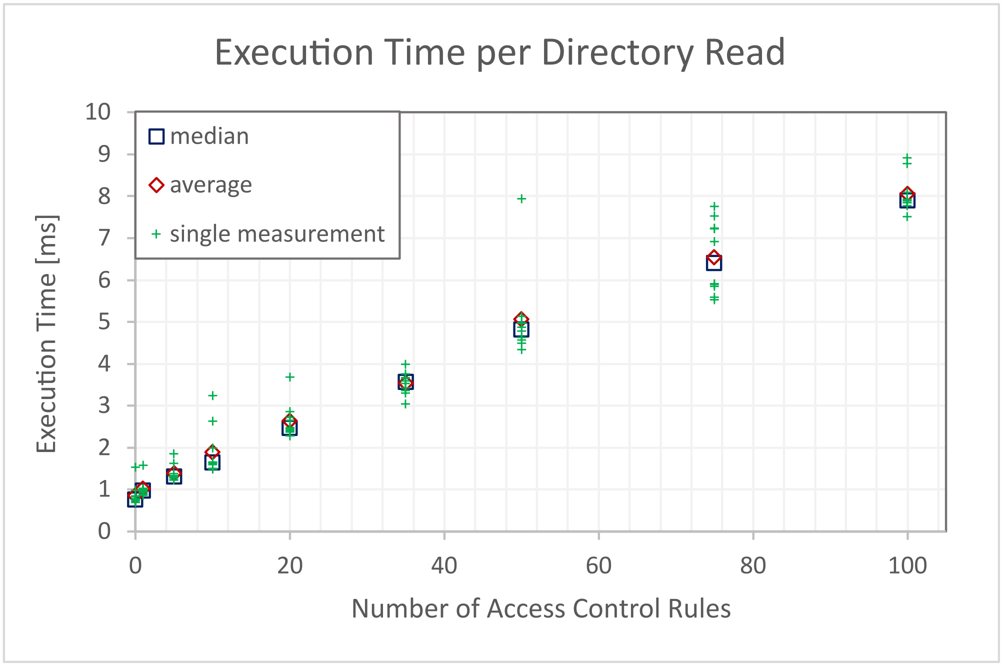


Figure 6.1: Benchmark Results for the Number of Access Control Rules

file. The files in this directory were listed 100 times per measurement and each measurement was repeated 10 times.

The benchmark used the following rule (in JavaScript notation):
`{rule: '!**/*.txt', except: {rule: '*/data/readme.txt'}}`

The results in figure 6.1 reveal an almost linear runtime complexity. Checking one such access control rule took 68µs on average. Since the rule checking involves a lot of string operations, these results are better than we expected.

6.3.2 Writers in a Collaboration Space

Reading the contents of a collaboration space requires querying the sub-graph of each participant. This is a rather resource-intensive operation. As a first step, it requires checking the pre-shared vertex of each user, which is implemented in the form of a graph database view. Each sub-graph is restricted to one Hypercore feed, which means that access control rules need to be checked for every vertex. The file status and metadata is read from the Hyperdrive filesystem for each file found by the graph query. That additional complexity already led to the expectation that the number of writers does not scale well.

This benchmark measured the runtime complexity of reading directory contents for a large number of participants in a collaboration space. For each measurement N users, in sequence, overwrote the same file. It is to be expected that N different files would have been even slower, because it would have required N additional file status read operations to different Hyperdrive filesystems. The files in this directory were listed 100 times per measurement and each measurement was repeated 10 times.

Figure 6.2 shows the results of the benchmark. These results indicate a polynomial runtime complexity. Reading the directory contents with $N = 50$ writers took almost a second, which is, in our opinion, a threshold where the load time

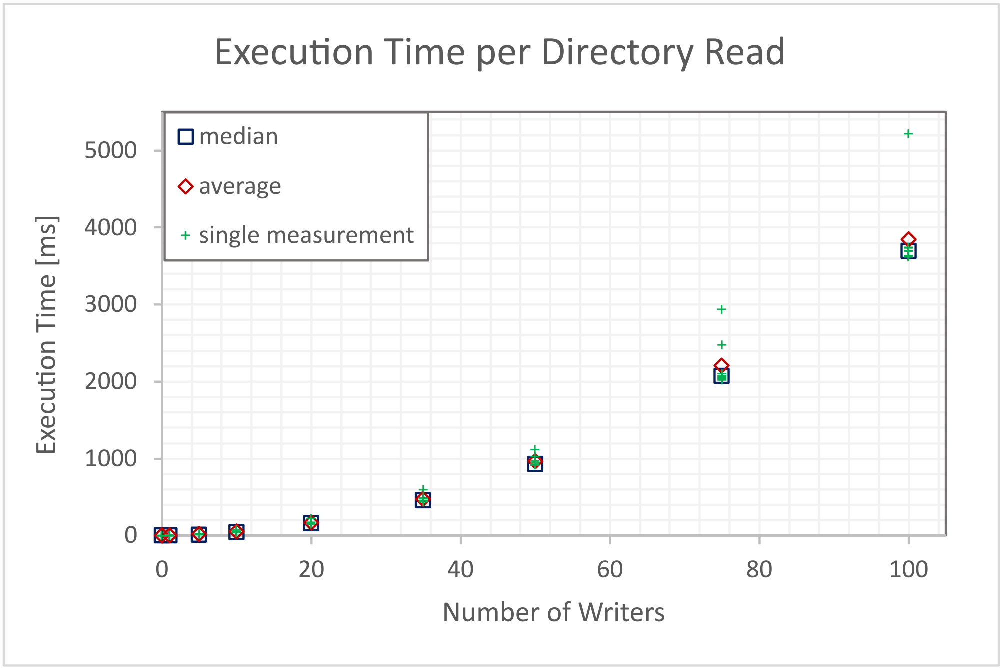


Figure 6.2: Benchmark Results for the Number of Writers in a Collaboration Space

might start to get annoying. Whether this is acceptable, depends on the type of application.

The results contain large single outliers. We suspect these outliers were caused by the software and hardware setup the benchmarks were executed on. There are many random variables that could cause such outliers, e.g. background processes or changes of the CPU clock due to thermal management.

6.3.3 Outbox Messages

Checking outbox messages follows a trial-and-error strategy for decrypting a sealed box, which utilizes public key cryptography. This benchmark measured the impact of a large number of messages in a user's outbox. For each measurement, N messages, addressed to user B, were placed into the outbox of user A. User B then read and decrypted all of the messages, 100 times per measurement. Each measurement was repeated 10 times.

The results in figure 6.3 indicate an average decryption duration of 160µs per message. Considering the previous benchmarks, we argue that the impact of the sealed boxes is almost negligible. It can be expected that other features cause serious bottlenecks before the Outbox becomes a problem. We expect that the results for addressing the messages to N different users would be very similar. Whether a trial-and-error attempt is successful or not should not make much of a difference.

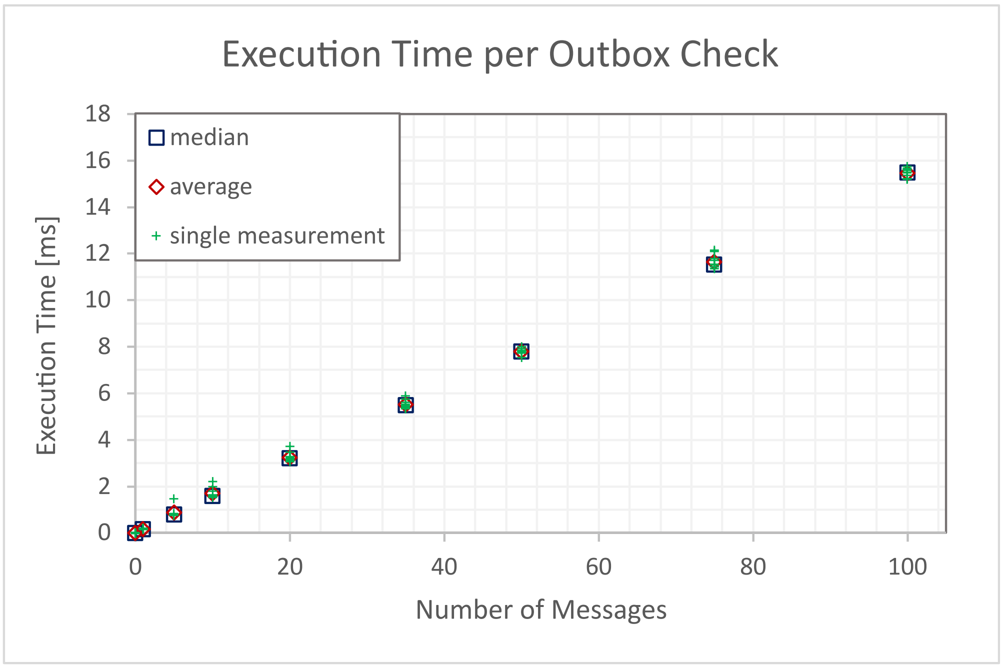


Figure 6.3: Benchmark Results for the Number of Outbox Messages

Chapter 7

Conclusion and Future Work

In this thesis, we analyzed different approaches to access control for P2P filesystems. The core motivation was to simplify building local-first software, based on which we defined a set of requirements for the desired system and identified challenges that needed to be solved. As the result of our work we presented our concept of an access control system for the P2P filesystem Hyperdrive. This concept was implemented and published as the NodeJS module CertaCrypt, accompanied by the demonstrator application CertaCrypt-Filemanager.

The cryptographic capability system based access control model features per-file and per-directory level read and write permission management, including the ability to revoke permissions for future changes. This is realized in the form of a graph, for key management, but also for other features, such as for user profiles, contacts and communication. The graph database built for this purpose turned out to be a useful, reusable tool for capability based access control that could potentially be used for applications other than filesystems as well. On the downside, unifying many features into such a general-use data structure vastly increases its complexity and is prone to mistakes during development.

Hyperdrive and the underlying Hypercore Protocol do not provide a user system, but that was crucial for our plan to simplify building local-first software. CertaCrypt provides a system that keeps track of a user's friends and other contacts, supports initial key exchange and implements a simple system for asynchronous communication. Trust is a difficult topic in P2P systems, for the demonstrator application we had to find a compromise between usability and security. In the case of trusting a previously unknown user that might actually be a malicious attempt to impersonate someone else, we ultimately chose usability over security to simplify the on-boarding of new users.

Due to a lack of resources, we had to leave a lot of useful features to future work and instead concentrate on those that were crucial for the access control system:

- The implemented write conflict resolution does not handle certain edge cases. CRDT-based write collision handling could solve many of those.
- Currently the C-style file handler API of Hyperdrive is not supported. The file handler API would allow random read access to file contents.
- In some cases it would be useful to have the ability to request write access to a collaboration space.
- In order to improve the availability of shared data, third parties can seed complete Hypercore feeds. This was a central requirement in the design phase. However, if a user wants the whole filesystem to be seeded, the replicate capabilities of the graph and those of the Hyperdrive need to be shared

separately. One possible future improvement, that would improve the usability a lot, would be a feature that allows sharing those replicate capabilities along with the information which Hypercore feed ranges need to be kept available.

- Another missing feature is the access to file versions. It is possible to check out the whole filesystem at a certain version, but CertaCrypt does not provide a feature to list the available versions of a file.
- Hypercore has a maximum block size of 8MB [SRC.8], which poses a hard limit for the size of a persisted vertex. This means the number of edges, and therefore the number of files in a directory, is limited. The exact limit depends on the data stored on the edges. This problem could be solved on the HyperObjects layer, e.g. by extending the object storage format in a way that allows splitting an object into multiple blocks.
- Another desirable feature, that could be implemented on the HyperObjects layer, is transaction log pruning. This would substantially reduce the required disk space by locally deleting old versions of the graph.
- In order to mitigate the threat of false attribution, child vertices that represent files or directories should refer to their parent vertices.
- The latest major releases of the utilized Hypercore Protocol modules were developed in parallel to our work. Upgrading to these improved versions would benefit our system's features and the overall level of security. Whether the multi-writer capabilities introduced with Hypercore 10 could be utilized for CertaCrypt, is another open question left for future work.

Many usability features of the CertaCrypt-Filemanager are left for future work as well. It only serves as a demonstrator for the technology, therefore it only provides what's absolutely necessary for that purpose. To make it a useful tool for real-world usage, it would require better OS integration and various other features that improve the user experience.

Overall, CertaCrypt provides a workable solution to the lack of access control in the P2P filesystem Hyperdrive and accomplishes its goal of simplifying the development of local-first software. The straight-forward implementation of the CertaCrypt-Filemanager supports that claim and demonstrates the capabilities of the access control system.

Bibliography

[1] Andre Alves Garzia, Val Lorentz, André Staltz. 2022. Scuttlebutt Protocol Guide. https://ssbc.github.io/scuttlebutt-protocol-guide.

[2] Phil Bagwell. 2001. Ideal Hash Trees. Technical report, October 2001. https://infoscience.epfl.ch/record/64398.

[3] Ammar Ayman Battah, Mohammad Moussa Madine, Hamad Alzaabi, Ibrar Yaqoob, Khaled Salah, and Raja Jayaraman. 2020. Blockchain-based multi-party authorization for accessing IPFS encrypted data. *IEEE Access*, 8, 196813–196825. DOI: 10.1109/ACCESS.2020.3034260.

[4] Ingmar Baumgart and Sebastian Mies. 2007. S/Kademlia: A practicable approach towards secure key-based routing. In *2007 International Conference on Parallel and Distributed Systems*, pp. 1–8. DOI: 10.1109/ICPADS.2007.4447808.

[5] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR*, arXiv:1407.3561 [cs.NI]. DOI: 10.48550/ARXIV.1407.3561.

[6] Daniel J. Bernstein. 2008. ChaCha, a variant of Salsa20. In *Workshop record of SASC* number 1. Volume 8. Lausanne, Switzerland, pp. 3–5. https://cr.yp.to/chacha/chacha-20080120.pdf.

[7] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography* (*PKC'06*). Springer, New York, NY, pp. 207–228. DOI: 10.1007/11745853_14.

[8] Daniel J. Bernstein. 2011. Extending the Salsa20 nonce. In *Workshop Record of Symmetric Key Encryption Workshop*. https://cr.yp.to/snuffle/xsalsa-20110204.pdf.

[9] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of cryptographic engineering*, 2, 2, 77–89. DOI: 10.1007/s13389-012-0027-1.

[10] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. 1992. The KeyKOS Nanokernel Architecture. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*. USENIX Association, USA, pp. 95–112. ISBN: 1880446421.

[11] Bryan Newbold, Stephen Whitmore, Mathias Buus. 2018. Dat Project Enhancement Proposals - DEP-0004: Hyperdb. https://github.com/dat-ecosystem-archive/DEPs/blob/30c4111d8847e41a312b0472adb9a2c093f5a69e/proposals/0004-hyperdb.md.

[12] Yi-Ruei Chen, Cheng-Kang Chu, Wen-Guey Tzeng, and Jianying Zhou. 2013. CloudHKA: A Cryptographic Approach for Hierarchical Access Control in Cloud Computing. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security* (*ACNS'13*). Springer, Berlin, Heidelberg, pp. 37–52. DOI: 10.1007/978-3-642-38980-1_3.

[13] Jack B. Dennis and Earl C. Van Horn. 1983. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 26, 1, 29–35. DOI: 10.1145/365230.365252.

[14] Colin J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10, 1, 56–66.

[15] Stefan Fixl. 2022. CertaCrypt Github Repository. Retrieved 2023-02-16 from https://github.com/fsteff/certacrypt.

[16] Frank Denis and other libsodium contributors. 2022. Libsodium Documentation. Retrieved 2023-02-17 from https://doc.libsodium.org/.

[17] Paul Frazee, Mathias Buus, Andrew Osheroff, and other contributors. 2022. Hypercore Protocol. https://web.archive.org/web/20221112231045/https://hypercore-protocol.org/.

[18] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer. 2006. Cryptree: A folder tree structure for cryptographic file systems. In *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*. IEEE, pp. 189–198. DOI: 10.1109/SRDS.2006.2.

[19] Amy Guy, Markus Sabadello, Drummond Reed, and Manu Sporny. 2021. Decentralized Identifiers (DIDs) v1.0. W3C Proposed Recommendation. W3C. https://www.w3.org/TR/2021/PR-did-core-20210803/.

[20] Anthony Harrington and Christian Jensen. 2003. Cryptographic Access Control in a Distributed File System. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies* (*SACMAT '03*). ACM, New York, NY, USA, pp. 158–165. DOI: 10.1145/775412.775432.

[21] Chris Hartgerink. 2019. Verified, Shared, Modular, and Provenance Based Research Communication with the Dat Protocol. *Publications*, 7, 2. DOI: 10.3390/publications7020040.

[22] Ian Preston, Kevin O'Dwyer. 2022. Peergos Book. Retrieved 2023-02-17 from https://book.peergos.org/.

[23] Holepunch Inc. 2022. Holepunch Docs. Retrieved 2023-02-17 from https://docs.holepunch.to/.

[24] Johnny Matthews and other contributors. 2022. IPFS Docs - Mutable File System (MFS). Retrieved 2023-02-17 from https://docs.ipfs.tech/concepts/file-systems/#mutable-file-system-mfs.

[25] Johnny Matthews and other contributors. 2022. What is Filecoin. Retrieved 2023-02-17 from https://docs.filecoin.io/about-filecoin/what-is-filecoin/.

[26] Jørgen Tvedt. 2017. Github Issue: Access Rights Scheme - Multiple keysets. Retrieved 2023-02-17 from https://github.com/hypercore-protocol/hyperdrive/issues/190.

[27] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (*Onward! 2019*). ACM, Athens, Greece, pp. 154–178. DOI: 10.1145/3359591.3359737.

[28] Martin Heidegger. 2018. Github Issue: Securing a Dat with a - additional - private key (password). Retrieved 2023-02-17 from https://github.com/dat-ecosystem-archive/datproject-discussions/issues/80.

[29] Petar Maymounkov and David Mazières. 2002. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Peer-to-Peer Systems. IPTPS '02* (*Lecture Notes in Computer Science (LNCS)*, volume 2429). Springer, Berlin, Heidelberg, pp. 53–65. DOI: 10.1007/3-540-45748-8_5.

[30] Ralph C. Merkle. 1980. Protocols for Public Key Cryptosystems. In *1980 IEEE Symposium on Security and Privacy*. IEEE, Oakland, CA, USA, pp. 122–134. DOI: 10.1109/SP.1980.10006.

[31] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. 2003. Providing authentication and integrity in outsourced databases using merkle hash trees. *UCI-SCONCE Technical Report*. https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkleodb.pdf.

[32] Mark Nadal. [n. d.] Conflict Resolution with guns. Retrieved 2022-08-23 from https://gun.eco/distributed/matters.html.

[33] Arvid Norberg. 2009. uTorrent Transport Protocol. Retrieved 2022-07-24 from http://bittorrent.org/beps/bep_0029.html.

[34] Maxwell Ogden, Karissa McKelvey, and Mathias Buus Madsen. 2017. Dat - Distributed Dataset Synchronization and Versioning. DOI: 10.31219/osf.io/nsv2c.

[35] Andrew Osheroff. 2020. Announcing Hyperdrive v10. *The Hypercore Protocol Blog*. Retrieved 2022-06-16 from https://blog.hypercore-protocol.org/posts/announcing-hyperdrive-10/.

[36] Paul Frazee, Bryan Newbold. 2019. Dat Project Enhancement Proposals - DEP-0010: Hypercore Wire Protocol. https://github.com/dat-ecosystem-archive/DEPs/blob/30c4111d8847e41a312b0472adb9a2c093f5a69e/proposals/0010-wire-protocol.md.

[37] Paul Frazee, Mathias Buus. 2018. Dat Project Enhancement Proposals - DEP-0002: Hypercore. https://github.com/dat-ecosystem-archive/DEPs/blob/30c4111d8847e41a312b0472adb9a2c093f5a69e/proposals/0002-hypercore.md.

[38] Trevor Perrin. 2018. The Noise protocol framework. https://noiseprotocol.org/noise.pdf.

[39] RangerMauve. 2018. Github Issue: Hiding data stored on registries. Retrieved 2023-02-17 from https://github.com/dat-ecosystem-archive/DEPs/issues/21.

[40] Danielle C. Robinson, Joe A. Hand, Mathias Buus Madsen, and Karissa R. McKelvey. 2018. The Dat Project, an open and decentralized research data tool. *Scientific data*, 5, 1, 1–4. DOI: 10.1038/sdata.2018.221.

[41] Sushmita Ruj, Milos Stojmenovic, and Amiya Nayak. 2014. Decentralized Access Control with Anonymous Authentication of Data Stored in Clouds. *IEEE Transactions on Parallel and Distributed Systems*, 25, 2, 384–394. DOI: 10.1109/TPDS.2013.38.

[42] Sander Pick, Carson Farmer, Aaron Sutula, Ignacio Hagopian, Irakli Gozalishvili, Andrew Hill. 2020. A protocol & event-sourced database for decentralized user-siloed data, Draft 1.6. Retrieved 2023-02-17 from https://docsend.com/view/gu3ywqi.

[43] Sehrish Shafeeq, Masoom Alam, and Abid Khan. 2019. Privacy aware decentralized access control system. *Future Generation Computer Systems*, 101, 420–433. DOI: 10.1016/j.future.2019.06.025.

[44] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (*SOSP '99*). ACM, Charleston, South Carolina, USA, pp. 170–185. DOI: 10.1145/319151.319163.

[45] Mathis Steichen, Beltran Fiz, Robert Norvill, Wazen Shbair, and Radu State. 2018. Blockchain-Based, Decentralized Access Control for IPFS. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, Halifax, NS, Canada, pp. 1499–1506. DOI: 10.1109/Cybermatics_2018.2018.00253.

[46] Romain Vaillant, Dimitrios Vasilas, Marc Shapiro, and Thuy Linh Nguyen. 2021. CRDTs for Truly Concurrent File Systems. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems* (*HotStorage '21*). ACM, Virtual, USA, pp. 35–41. DOI: 10.1145/3465332.3470872.

[47] Peter van Hardenberg and Martin Kleppmann. 2020. PushPin: Towards Production-Quality Peer-to-Peer Collaboration. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data* (*PaPoC '20*), Article 10. ACM, Heraklion, Greece, pp. 1–10. DOI: 10.1145/3380787.3393683.

[48] Werner Vogels. 2009. Eventually Consistent. *Communications of the ACM*, 52, 1, 40–44. DOI: 10.1145/1435417.1435432.

[49] Zooko Wilcox-O'Hearn and Brian Warner. 2008. Tahoe: The Least-Authority Filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability* (*StorageSS '08*). ACM, Alexandria, Virginia, USA, pp. 21–26. DOI: 10.1145/1456469.1456474.

[50] Shuhua Wu and Kefei Chen. 2012. An efficient key-management scheme for hierarchical access control in e-medicine system. *Journal of Medical Systems*, 36, 4, 2325–2337. DOI: 10.1007/s10916-011-9700-7.

[51] Yoshua Wuyts. 2019. Dat Project Enhancement Proposals - DEP-0009: Hypercore SLEEP Headers. https://github.com/dat-ecosystem-archive/DEPs/blob/30c4111d8847e41a312b0472adb9a2c093f5a69e/proposals/0009-sleep-headers.md.

[52] Ronghua Xu, Yu Chen, Erik Blasch, and Genshe Chen. 2018. BlendCAC: A Smart Contract Enabled Decentralized Capability-Based Access Control Mechanism for the IoT. *Computers*, 7, 3. DOI: 10.3390/computers7030039.

Project Code References

Ref.	NPM Package Name	Description	Git Repository URL (Version Commit)	Version
[SRC.1]	@certacrypt/certacrypt	CertaCrypt main module	https://github.com/fsteff/certacrypt/tree/f3914a2a538d9a5a22b30f93c5f239df5a79f301	1.0.0
[SRC.2]	hyperobjects	Transaction log based object store	https://github.com/fsteff/hyperobjects/tree/bfc42d3beb89d7291c9fa6a664c28b024054ca2b	1.0.1
[SRC.3]	@certacrypt/hyper-graphdb	Graph database	https://github.com/fsteff/hyper-graphdb/tree/638d1f6ab7c769d2a8adfb275b9ddeef72fba207	1.0.1
[SRC.4]	@certacrypt/certacrypt-graph	Access control extension for hyper-graphdb	https://github.com/fsteff/certacrypt-graph/tree/79ae559b383f2bb6c44d64ffa76bba34aba67ea5	1.0.0
[SRC.5]	@certacrypt/certacrypt-crypto	Crypto utilities for CertaCrypt	https://github.com/fsteff/certacrypt-crypto/tree/a974d9e12090d26f1fc19858d6ce4109867fc9d2	1.0.0
[SRC.6]	certacrypt-filemanager	Demonstrator application for CertaCrypt	https://github.com/fsteff/certacrypt-filemanager/tree/639ef9436bc11d8eaee49084f0112ec7632395db	1.0.0
[SRC.7]	hyperpubsub	PubSub system as a Hypercore Protocol extension	https://github.com/fsteff/hyperpubsub/tree/95552c70bfd932f62049c87cd83c9c55243599b4	1.2.4

Hypercore-Protocol Code References

Ref.	NPM Package Name	Description	Git Repository URL (Version Commit)	Version
[SRC.8]	hypercore	Main Hypercore module	https://github.com/holepunchto/hypercore/tree/74fd5f5712188d11b885023342819a635fc303ec	9.12.0
[SRC.9]	hypercore	Hypercore version 10	https://github.com/holepunchto/hypercore/tree/fc541afec9c3c617f236b77e65b4cda8c7ab8197	10.7.0
[SRC.10]	hypercore-protocol	High-level implementation of the Hypercore replication protocol	https://github.com/hypercore-protocol/hypercore-protocol/tree/c299117fbcbe7100adf442f30768ec4860cacda7	8.0.7
[SRC.11]	hypercore-crypto	Cryptography utilities used for Hypercore	https://github.com/mafintosh/hypercore-crypto/tree/b6e2d99f9b6634e7d4e3d36b5f6502a88d1eee50	2.3.2
[SRC.12]	simple-hypercore-protocol	Low-level implementation of the Hypercore replication protocol	https://github.com/mafintosh/simple-hypercore-protocol/tree/972740e8e97779bc9464361fbfa6211aca3b3ec2	2.1.2
[SRC.13]	simple-handshake	Handshake using the NOISE framework	https://github.com/emilbayes/simple-handshake/tree/7ba3027721b5b380e8a8d98aa7a9cdf0bf1434de	3.0.0
[SRC.14]	noise-protocol	Javascript implementation of the Noise Protocol Framework based on libsodium [SRC.15]	https://github.com/emilbayes/noise-protocol/tree/331566df32160ce8a7fdfab9d4556b00301af999	3.0.1
[SRC.15]	sodium-universal	Universal wrapper for sodium-javascript and sodium-native working in NodeJS and the Browser	https://github.com/sodium-friends/sodium-universal/tree/69753b149513fb1688ad44fcc245cbee9616af06	3.1.0
[SRC.16]	hyperdrive	Main Hyperdrive module	https://github.com/hypercore-protocol/hyperdrive/tree/220de8818064e01a6fa517b65d108a1e192bd969	10.21.0
[SRC.17]	hypertrie	Rolling hash array mapped trie to index key/value data on top of Hypercore	https://github.com/hypercore-protocol/hypertrie/tree/37dc7925d6f839b966528f62bdaaa5d2e99acccb	5.1.3

Ref.	NPM Package Name	Version
	Description	
	Git Repository URL (Version Commit)	
[SRC.18]	mountable-hypertrie	2.8.0
	A Hypertrie [SRC.17] wrapper that supports mounting of other Hypertries	
	https://github.com/andrewosh/mountable-hypertrie/tree/f4a64f98fe86fc5725d6ba9ebf3d4b45d4b3ad0b	
[SRC.19]	hypercore-bytestream	1.0.12
	A Readable stream wrapper around Hypercore that supports reading byte ranges. Used for Hyperdrive [SRC.16]	
	https://github.com/andrewosh/hypercore-byte-stream/tree/9b648f50806139dba7f76a8c8317911172c0d388	
[SRC.20]	random-access-storage	1.4.3
	For implementing random-access-storage interface compliant modules	
	https://github.com/random-access-storage/random-access-storage/tree/50dd02fd07d5d1690a3346684f8aac74dd76d017	
[SRC.21]	random-access-memory	6.1.0
	Virtual in-memory storage	
	https://github.com/random-access-storage/random-access-memory/tree/9e504ea2637ab99db6120c7ee54bbf1bfad47d42	
[SRC.22]	random-access-s3	0.0.2
	A random access interface for AWS s3 buckets (read only)	
	https://github.com/random-access-storage/random-access-s3/tree/1e38579b7b152111651930d5effa0933243a0524	
[SRC.23]	hyperdrive-next	11.0.0-alpha.10
	Hyperdrive 11 alpha preview based on Hypercore 10	
	https://github.com/holepunchto/hyperdrive-next/tree/fbd973785845a6c3ba496c5b47f71b1e9b24ac6c	
[SRC.24]	hyperswarm	2.15.3
	Main Hyperswarm module	
	https://github.com/holepunchto/hyperswarm/tree/924f56ae2781f1c605916e8e070f474ac7fff0f0	
[SRC.25]	@hyperswarm/discovery	2.0.1
	Hyperswarm discovery using the DHT and mDNS	
	https://github.com/hyperswarm/discovery/tree/e9ca6a7a4ddd099c71eb68797793322a116b2159	
[SRC.26]	@hyperswarm/dht	4.0.1
	The DHT powering Hyperswarm	
	https://github.com/holepunchto/hyperswarm-dht/tree/c8bbe643dac374d9c7cf92d723b732a980c564bd	
[SRC.27]	@hyperswarm/dht	6.5.0
	The DHT powering Hyperswarm, latest version	
	https://github.com/holepunchto/hyperswarm-dht/tree/40653a29f5901fe3668a4893959a1b89490dd2f7	
[SRC.28]	@hyperswarm/network	2.1.0
	Low-level networking parts of Hyperswarm	
	https://github.com/hyperswarm/network/tree/12a211462c1f4ddcab8ee73398f6144026a08c09	

Ref.	NPM Package Name	Description	Git Repository URL (Version Commit)	Version
[SRC.29]	dht-rpc	RPC calls over a Kademlia based DHT, used by [SRC.26]	https://github.com/mafintosh/dht-rpc/tree/19a0df46cd63ec0ddfff16b7ffab77d384e0c4db	4.9.6
[SRC.30]	dht-rpc	RPC calls over a Kademlia based DHT, latest version	https://github.com/mafintosh/dht-rpc/tree/949d146db4cb79bb6baf8ca08716b141bc79f060	6.3.0
[SRC.31]	udx-native	libudx - reliable, multiplex, and congestion controlled streams over udp (pre-alpha WIP version)	https://github.com/hyperswarm/libudx/tree/eca01e45e116d326c5173ee32e9d80f9662c81c4	1.4.0
[SRC.32]	corestore	Canonical implementation of the corestore interface	https://github.com/hypercore-protocol/corestore/tree/87d11b3d2bec28193797abc795938ea4d17d7c51	5.8.2
[SRC.33]	hyperspace	Lightweight server that provides remote access to Hypercores and a Hyperswarm instance and exposes a simple RPC interface	https://github.com/hypercore-protocol/hyperspace/tree/26d6d36f3d3f9d6ca1269994af5a2ddf9096c583	3.19.0
[SRC.34]	hyperbee	B-tree running on a Hypercore	https://github.com/holepunchto/hyperbee/tree/295f465c85970be862d0715cfb429d091076a5b0	2.4.2
[SRC.35]	autobase	Rebase multiple causally-linked Hypercores into a single, linearized Hypercore	https://github.com/holepunchto/autobase/tree/7719d9a87fd191563044cb14468340ce42b6d4c7	1.0.0-alpha.8